University of California
Santa Barbara

Data Management Solutions for Tackling Big Data Variety

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Vaibhav Arora

Committee in charge:

Professor Divyakant Agrawal, Co-Chair
Professor Amr El Abbadi, Co-Chair
Professor Chandra Krintz

March 2018

The Dissertation of Vaibhav Arora is approved.

Professor Chandra Krintz

Professor Amr El Abbadi, Committee Co-Chair

Professor Divyakant Agrawal, Committee Co-Chair

March 2018

Data Management Solutions for Tackling Big Data Variety

by

Vaibhav Arora

Acknowledgements

This dissertation would not have been possible without the support, company and contribution of many people. Words alone are not enough to express the value of their part in this work.

First and foremost, I am deeply indebted to my advisors Divyakant Agrawal and Amr El Abbadi, for their continuous support, mentorship and guidance throughout the PhD. They have taught me about abstracting and formulating research problems, shaping ideas and solutions, and putting them coherently on paper. Both of them are very approachable, and have fostered a great working environment for their students. They have the ability to sense the time of distress for their students, and have provided me encouragement and support during such hard times. Their long professional partnership is inspiring and very helpful for their co-advised students like me. Divy and Amr get great joy from the success of their students and I am very thankful to them for their help and advice in deciding the next steps after the PhD.

I would also like to acknowledge Chandra Krintz, for being on my doctoral committee. Chandra was also my advisor in the first year of grad school, when I came in as a Masters student. I am grateful to her for encouraging and introducing me to the research world, and stressing the need of deeply analyzing and questioning experimental results. I am also appreciative of Janet Kayfetz for her inputs on technical writing and presentations, which have helped me better express my thoughts and ideas.

The various projects included in this thesis have had different collaborators, who have played an invaluable part in them. Faisal Nawab has been my collaborator on several projects, and has provided deep insights and suggestions. Tanuj Mittal, Ravi Kumar, Sujaya Maiyya, Mohammad Amiri, Xun Xue, Zhujianfeng, Zhiyanan have also provided invaluable contributions on different projects. I am also grateful to mentors during my

internships at Amazon, HP Labs and Microsoft Research. The internships have been a great learning experience and have given a chance to get acquainted with different perspectives and approaches to problem solving, and diverse working styles.

A lot of my time in the last few years has been spent in the Distributed Systems Lab (DSL), and I have been fortunate to share it with a great set of labmates. Faisal, Cetin, Aaron, Hatem, Theodore, Xiaofei, Siva, Victor, Sujaya and Mohammad have been a constant source of encouragement, motivation, suggestions and feedback. Its been a pleasure to share successes and failures, and to spend time, socialize and have fun with them both inside and outside the lab.

I am thankful for the friendships kindled during my time as a graduate student at UCSB. Apart from my lab mates, Abhishek, Anand, Varad, Deepak, Manish, Stef, Stratos, Alex, Nevena, Deeksha are some of the friends I have made in Santa Barbara. It has been wonderful to hangout, and enjoy life in Santa Barbara with them, and also have their support and suggestions during the times of need. Some of my other old and new friends in different parts of the world, have also been there for me, and provided me with wonderful company.

My family has been a source of strength, support and good times. I am indebted to my parents for their love and affection, and for inculcating the value of education and hard work. My brother, Vipul and sisters, Alka and Shipra, have always encouraged and motivated me. I am especially grateful to my parents and siblings for their guidance during the early formative years. Rajeev, Preeti, Gauri, Vanshu, Kritin, Akshaj, Arushi, Aditya and Rhea, have been a part of some happy moments over the years. I am thankful to my girlfriend, Elizabeth, for providing me with constant support in different aspects of life, during the last few years of the PhD. She has heard numerous paper reject stories, and has been by my side and kept me going, and helped achieve a balance (whatever the PhD life can afford) and well-being in life outside work.

Many other people have played a part in my overall quality of life at Santa Barbara. The interactions with the graduate and undergraduate student community, faculty and staff at UCSB have provided me an enriching experience. I have also discovered the joy of Salsa dancing during the last few years, and am appreciative of the Salsa community, for playing a part in me gaining balance outside the academic world.

Conducting research also needs material support, and I want to acknowledge National Science Foundation, NEC and Huawei for their research grants, and Amazon Web Services for giving credits for performing experiments.

Curriculum Vitæ
Vaibhav Arora

Education

2018	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2016	M.S. in Computer Science, University of California, Santa Barbara.
2009	B.Tech. in Computer Science, National Institute of Technology, Tiruchirappalli.

Publications

Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi: Janus: A Hybrid Scalable Multi-Representation Cloud Datastore. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2018: 30(4), 689-702.

Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi: Typhon: Consistency Semantics for Multi-Representation Data Processing. IEEE Cloud Computing (CLOUD), 2017: 648-655.

Vaibhav Arora, Tanuj Mittal, Divyakant Agrawal, Amr El Abbadi, Xun Xue, Zhiyanan Zhiyanan, Zhujianfeng Zhujianfeng. Leader or Majority: Why have one when you can have both? Improving Read Scalability in Raft-like consensus protocols. Usenix Hot Topics in Cloud Computing (HotCloud), 2017.

Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi: Multi-representation based data processing architecture for IoT applications. IEEE International Conference on Distributed Computing Systems (ICDCS), 2017: 2234-2239.

Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, Amr El Abbadi: Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores. ACM International Conference on Management of Data (SIGMOD), 2015: 1279-1294.

Aaron J Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, Amr El Abbadi: Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. ACM International Conference on Management of Data (SIGMOD), 2015: 299-313.

Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, Amr El Abbadi: Chariots: A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments. Extending Database Technology (EDBT), 2015: 13-24.

Divyakant Agrawal, Amr El Abbadi, Vaibhav Arora, Ceren Budak, Theodore Georgiou, Hatem A. Mahmoud, Faisal Nawab, Cetin Sahin, Shiyuan Wang: Mind your Ps and Vs: A perspective on the challenges of big data management and privacy concerns. BigComp 2015: 1-6.

Hatem Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi: MaaT: Effective and Scalable Coordination of Distributed Transactions in the Cloud. Very Large Data Base Endowment (VLDB), 2014: 7 (5), 329-340.

Chris Bunch, Vaibhav Arora, Navraj Chohan, Chandra Krintz, Shashank Hegde and Ankit Srivastava. A Pluggable Autoscaling Service for Open Cloud PaaS Systems. Utility and Cloud Computing (UCC), 2012: 191-194.

In Submission. Vaibhav Arora, Ravi Kumar Suresh Babu, Suajaya Maiyya, Divyakant Agrawal, Amr El Abbadi, Xun Xue, Zhiyanan Zhiyanan, Zhujianfeng Zhujianfeng. Dynamic Timestamp Allocation for Reducing Transaction Aborts.

Experience

Microsoft Research. Seattle, WA, USA — June 2015 - September 2015
Research Intern

HP Labs. Palo Alto, CA, USA — June 2014 - September 2014
Research Intern

Amazon. Seattle, WA, USA — June 2013 - September 2013
Software Development Engineering (SDE) Intern

Amazon. Seattle, WA, USA — June 2012 - September 2013
Software Development Engineering (SDE) Intern

Yahoo! R & D. Seattle, WA, USA — June 2009 - Aug 2011
Software Engineer

Abstract

Data Management Solutions for Tackling Big Data Variety

by

Vaibhav Arora

Variety is one of the three defining characteristics of Big Data; the others being Volume and Velocity. There are several aspects of this data variety: diversity in data formats (text, video, audio) and structure (relational, graph etc), variety in access methodologies (OLTP, OLAP), and distribution heterogeneity within the workloads (read-heavy, high contention). Data management solutions for modern-day applications need to tackle this variety.

This dissertation provides an understanding of the challenges associated with the different elements of variety, and proposes several solutions for efficiently handling its various aspects. First, the dissertation studies the challenges related to variety in data structure and access methodologies, and the resultant heterogeneity at the data infrastructure level. Applications now employ several data-processing engines with different underlying representations, like row, column, graph etc., to process their data. We propose Janus, which introduces a novel data-movement pipeline, which enables the use of different representations to support both high throughput of transactions and diverse analytics, while still ensuring consistent real-time analytics in a scale-out setting. Janus partitions the data at different representations, and allows distributed transactions and diverse partitioning strategies at the representations. Then, we propose Typhon and Cerberus, which define and enforce consistency semantics for application data spread across representations. Second, this dissertation proposes solutions for handling distribution heterogeneity within the workloads. Workloads can have have skewed distribution

in terms of operation-type, data access or temporal variation. We propose strongly-consistent quorum reads for Raft-like consensus protocols, which can be utilized to scale read-heavy workloads. For supporting high contention transaction workloads, we integrate an existing dynamic timestamp allocation based concurrency control mechanism in a distributed OLTP setting, and analyze its performance. Third, we study IoT applications, which have to deal with both physical heterogeneity of the sensors, as well as diverse data-processing demands. We propose a multi-representation based architecture catering to IoT applications, and also present the initial design of M-stream, a computation framework for enabling integration and monitoring of uncertain data from multiple sensors. Through analysis, illustrative examples and extensive evaluation of the proposed protocols, this dissertation demonstrates that the proposed solutions can be employed for efficiently handling the different aspects of variety of data-intensive applications.

Contents

List of Figures

Chapter 1

Introduction

The last decade has seen a huge growth in the amount of data being generated and processed. This has lead to the emergence of a plethora of web applications with diverse demands. E-commerce, online banking, retail, social networking and IoT (Internet of Things) applications are some such examples. Such applications have disparate data management needs. Different methods have been proposed to classify the data processing demands of modern-day applications. One of the ways to classify the data processing characteristics of such applications has been the 3V's: Velocity, Volume and Variety [1].

1.1 Data Variety

Variety is one of the most challenging aspects of the Big Data ecosystem. *Variety* or *Heterogeneity* comes in many diverse aspects. Typically, the term Variety is used to refer to the different formats of data content, for example, text, image, videos etc. However, in addition to the variability of data formats, there are other important aspects of *variety*.

Data is often represented in different formats. Traditionally, data can be present as structured, semi-structured or unstructured. In the unstructured case, data is represented

simply by its content, i.e., text, image, video etc. However, structure on the data, for example, as relational, key-value, graph, RDF etc., highlights significant relationships that are intrinsic to the sub-components of the data.

In addition to the variety of data representation, there is a variety of access methodologies, for example OLTP (Online Transactional Processing), OLAP (Online Analytical Processing), Graph Based, Stream or Event-based processing etc. Based on the characteristics of the application, it might need to support a subset or all of various types of updates (transactional, event-based) and diverse reads and analytical operations (real-time analytics, batch processing). To consider the example of variety of accesses needed by applications, consider an E-commerce application. Inventory management, purchase management, and operations looking-up individual items have transactional demands. Such applications also want to execute aggregate operations to analyze business performance, like finding the number of products sold per regions, or by user age, which are characteristics of OLAP workloads. Furthermore, E-commerce applications also want to recommend products to users based on likes and dislikes of other similar users. Such recommendation queries might be executed on a graph with user and product connections. Social networking applications have diverse access requirements as well. Status updates can be categorized as continuous events of a data stream, adding and removing friends has transactional requirements, and user behavior is categorized by running analytical queries. Applications need to efficiently support such diverse workloads.

Another aspect of variety is the distribution heterogeneity within the various workloads (OLTP, OLAP, Graph-based) of data-intensive applications. Many such workloads are skewed in terms of one or more of the following attributes: types of operations (reads or writes), distribution over data items, and temporal characteristics. Workloads of social networks like Facebook are highly skewed in the favor of reads [2]. Additionally, access distribution over the data items is highly non-uniform for OLTP workloads, re-

sulting in *hot-spots* over certain tuples or ranges of data items [3]. For example, a high volume of operations (40-60%) on the New York Stock Exchange (NYSE) occur on a small fraction of stocks (around 1%). Another example is a social networking application such as Twitter, where celebrities can have million of followers, and such tuples might be accessed orders of magnitude more than an average user [4]. Many workloads also have time-varying skews. Some of these skews are predictable, whereas some of them are sudden. E-commerce websites like Amazon face a large spike in requests during the Holiday season. Furthermore, many web applications experience unexpected sudden surges in load due to a very high rate of user growth [5] or unforeseen world events [6].

An emerging class of applications which illustrates and stresses the variety aspect of data processing architectures, are IoT (Internet of Things) applications. IoT applications are growing at a rapid rate and will have a huge impact on our future. Smart cars, smart cities, weather monitors, smart farms and fitness and health tracking wearable devices are some examples of internet connected devices, which continuously transmit data points from connected sensors. IoT applications continuously collect and process the data received from these devices. These applications need to support both high frequency data ingestion as well as real-time analytics. Monitoring the weather in disaster prone areas, routing smart cars, tracking health data like heart rate are some scenarios where insights into the data are needed in real-time. In all the examples mentioned, real-time analytics must be considered while ingesting a high incoming rate of data. The data ingestion and the analytical processing demands of IoT applications are also diverse in nature. Analytical queries might be online queries (pre-defined aggregates over a certain attribute like temperature in a weather-related app), graph-processing requests (calculating activity in a connected community of a user in a motion tracking app) or offline batch processing queries (models for predicting the long-term likelihood of droughts, cyclones etc., executing over months/years of weather data). Data ingestion

might be in the form of continuous independent events or values from multiple related sensors which might need to be ingested atomically.

Another aspect of the IoT ecosystem is the physical heterogeneity of the sensors. IoT applications may have to handle data from a diverse set of sensors, which capture the attributes of their corresponding physical settings, and might want to integrate such data to gain more value and insights. For example, information from diverse medical sensors calculating blood pressure, heart rate etc. can be combined for efficiently monitoring and administering drugs to the patients [7].

Modern-day application data and corresponding workloads have been characterized by diverse data formats, structure and different access methodologies, as well as heterogeneity with the workloads. The variety has had a huge impact on the data management ecosystem and poses significant challenges that need to be handled by the data-processing architectures employed by the applications.

1.2 Impact of Data Variety

1.2.1 Multi-Engine Architecture

The tremendous variety of data processing demands has led to the fall of "one-size-fits-all" paradigm [8]. Applications now use different systems for different use cases rather than using a single relational database for all of their demands. Specialized OLTP engines like VoltDb [9] and H-Store [10], Distributed Key-value stores like BigTable [11] and Dynamo [12], OLAP targeted systems such as Vertica [13], MonetDB [14, 15] and C-Store [16], Graph databases such as Neo4j [17], and Stream processing engines like Storm [18] and Heron [19] are some examples of the systems which have been used in addition to the traditional relational database engines [20, 21, 22, 23].

Maintaining separate systems for different needs such as transactional processing, online analytics and graph processing operations helps to cater to specific performance characteristics. Many of the diverse systems mentioned above use different data representations to optimize for different workload types. Traditional OLTP databases like System R [20] employ row-based storage. Row-based storage can help in optimizing for OLTP workloads, by taking advantage of sequential I/O, since most of the operations touch many attributes for a small number of tuples [24]. Column-based storage [25] optimizes for OLAP workloads, which have queries which access a small number of attributes for a large number of tuples. Storing data column wise is more advantageous in such settings, as it results in cost savings through the ability to perform block iterations, compression and late materialization [24]. Vertica [26] and MonetDB [15] are two examples of systems employing column-based storage for targeting OLAP workloads. Storing the data in graph format as nodes and edges, as done in Neo4j [17], can be advantageous for certain networked datasets like social networks and movie databases. If we consider a movie database, with actors and movies being represented as nodes and edges representing the "acted-in" relationship, a query for traversing all the movies in which two particular co-actors have appeared, can be faster with graph representation. By using graph storage, we can avoid multiple joins, which will be needed in a relational database and instead have direct access to the relationships through the edges.

We refer to the architecture of using different data processing systems as *multi-engine* or *multi-representation* architecture. Multi-representation architecture can delegate various workloads to the most suitable data processing engine, employing the most suited underlying representation for the workload. However, such an architecture has many challenges.

Support for real-time analytics. First, such an architecture is not amenable to real-time analytics. Separate ETL (Extract, Transform and Load) processes and CDC (Change

Data Capture) systems [27] are needed to transfer data from the operational data processing engines, optimized for supporting updates to various analytical engines. If the updates are transactional, they need to be consistently merged at the analytical engines. This transfer introduces a delay before the updates on the data are reflected in the analytical results.

Consistency across data processing engines. Second, most of the data processing engines enforce consistency at a granularity of a single datastore. This can lead to consistency violations for the application user, when the application data is spread and accessed across multiple data processing engines or representations. To illustrate inconsistent semantics in applications accessing data across multiple representations, consider a social networking application. The social network consists of users, with attributes and a list of friends. The user attributes are stored in a key-value store and friend relationships are stored in a graph database, to take advantage of the different characteristics of user attribute data and the friendship graph. Now consider two users: Alice and Bob. The specifications of the application only permit the friends of a user to access a user's personal information stored in the key-value store. For example, since Alice and Bob are friends, Bob has access to Alice's phone number. Suppose, Alice wants to change her phone number and does not want Bob to access it. Alice deletes Bob from her friends-list and subsequently, she changes her phone number. Any request by Bob to read Alice's phone number should not be able to access Alice's new phone number. Providing even the strongest guarantees individually at the key-value store and the graph database does not provide this consistency guarantee across the different systems, and concurrent read requests by Bob in the scenario, can lead to consistency violations.

Choosing between Data Processing Engines. Third, applications have to chose between different data-processing engines for executing different workloads. This also requires the application developer to choose engines which satisfy the data management requirements

of an application workload or query. This is not ideal as a developer might not be best suited to make such a decision.

For tackling the variety of multiple data formats, structures and access methodologies more efficiently, challenges posed by multi-engine architecture need to be addressed. In this dissertation, we propose solutions which are able to provide support for real-time analytics and understand and address consistency violations, while still benefiting from the use of different data-processing engines backed by diverse representations.

1.2.2 Workload Heterogeneity

Workload heterogeneity in the form of skewed operation type (reads vs writes ratio), data access skew and temporal variation has a big impact on the performance of a data management system. The impact of workload heterogeneity is especially exacerbated in large-scale systems.

Consider an application deployed at scale and supporting OLTP like workloads with reads and write operations supported over a set of data items. Data management architectures typically employ replication in this environment, for supporting both fault-tolerance and performance concerns. Consensus protocols [28, 29, 30] are used to synchronize between these replicas, and allow the replicas to act as one coherent group. Many consensus protocols like Multi-Paxos [29] and Raft [28], employ leader-based approaches for achieving consensus. Write operations are written to a majority of replicas before committing, whereas read operations go through the leader to ensure linearizability. This ensures that every read operations returns the latest value of the data item. Such an approach does well under uniform workloads, but has performance issues when operations are skewed. In the presence of read-heavy workloads, the system might have bottlenecks at the leader, resulting in poor performance. Furthermore, since reads go through the

leader, non-leader replicas are not utilized very efficiently for these workloads. Although these replicas are ready to fail-over in presence of crashes, they have low-utilization in failure-free scenarios.

For supporting transactions in such a setting, distributed database architectures [31, 32, 33] typically build a concurrency-control and distributed commitment mechanism over the consensus protocol. Distributed databases often employ either pessimistic schemes using timestamp ordering [33], or optimistic concurrency-control (OCC) based protocols [32], for the concurrency-control mechanism. These schemes are lock-free and do not block transactions due to locking, which can poorly affect performance in distributed environments. However, for these lock-free schemes, the performance of transactional workloads is highly affected by data access skews. Protocols like pessimistic timestamp ordering use statically allocated timestamps to order transactions at commit, which can lead to a lot of ordering conflicts in the presence of conflicting transactions. In the presence of hot-spots/high contention, a number of transactions have to be aborted, leading to a drop in throughout and wasteful work.

Due to the prevalence of workload distribution variety, in the form of both operation-type and data access skew, it is imperative for distributed database architectures to employ techniques to handle such workload non-uniformity. In this dissertation, we propose solutions to efficiently tackle read-heavy workloads and data access skew respectively.

1.3 Dissertation Overview

We now overview the work proposed in the dissertation. This dissertation analyzes the fundamental challenges posed by the *variety* of data and its corresponding processing needs in modern applications. We build systems and propose solutions which address the different aspects of data variety. Figure 1.1 illustrates the different components of the

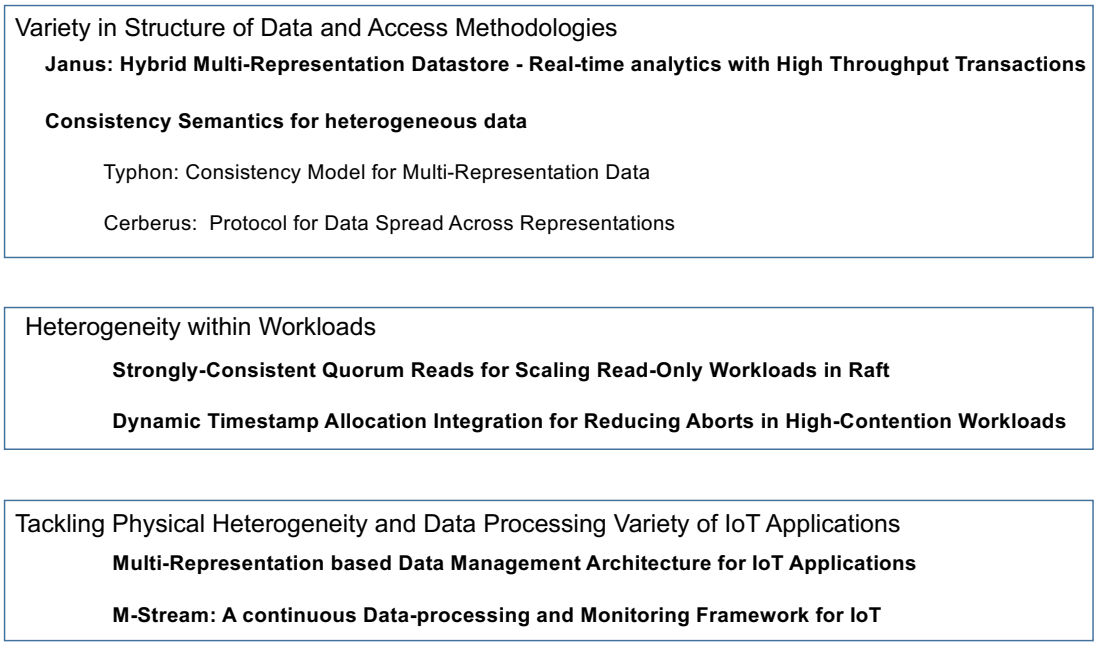


Figure 1.1: An Overview of the dissertation work

proposed work. These components offer solutions for different challenges of variety faced by data-intensive applications. Next, an overview of different parts of the dissertation is presented.

1.3.1 Variety in Data Structure and Access Methodologies

In the first part of the dissertation, we propose solutions to handle the different challenges posed by variety of data formats, structure and access methodologies. A hybrid scalable multi-representation cloud datastore, **Janus** [34] is designed to enable real-time analytics, while supporting a high frequency of updates. Janus can store the data in multiple representations, and provide a high throughout of transactional operations, while enabling real-time analytics in a large-scale setting. One copy of the data

is stored in a write-oriented representation, whereas other copies are marked read-only. Janus supports large-scale data by partitioning, and both the write-oriented representation and the read-oriented representation are partitioned. Janus proposes a novel *data movement pipeline*, which ships data from write-oriented representations to analytics or read-oriented representations. The data-movement pipeline in Janus is closely tied to the concurrency-control mechanism at the write-oriented representation. It captures data in batches and employs a graph-based dependency management algorithm to ensure that all the transactional updates are consistently merged at the read-oriented representation in near real-time. Janus ensures that any analytical query will reflect the same serialization order of transactions as observed on the transactional data, and partial affect of any transaction would not be observed. Janus ensures this guarantee in the presence of distributed transactions and diverse partitioning strategies at the representations. An evaluation performed on AWS Cloud Platform demonstrates that Janus is capable of supporting real-time analytics and high transaction throughput, while scaling-out.

The problem of consistency in a multi-representation architecture is then studied. We first propose a multi-representation data processing framework, **Typhon** [35], which formally defines consistency semantics in a setting where application data is spread across multiple data representations. The consistency model defines *implicit causal dependencies* which capture the logical order intended by the application user. The consistency model in Typhon integrates with the traditional conflict-graph model in databases. An extra set of edges is added to the conflict graph model to capture the dependencies of operations performed across data representations, which enables formally defining and identifying consistency violations in a multi-representational setting. A protocol, named **Cerberus** [35], is then proposed, which achieves the consistency semantics defined in Typhon's multi-representation model. The guarantees provided by Cerberus are formally proved. Experimental evaluation shows that Cerberus presents a lightweight and scal-

able mechanism, with relatively small overhead when compared to a solution providing no consistency guarantees for operations accessing data across representations. The proposed protocol enables applications to utilize diverse representations to store different data, while not violating consistency for the users, like in the social networking example presented above.

1.3.2 Workload Heterogeneity

In the second part, solutions are proposed to tackle the distribution heterogeneity of OLTP workloads, executed by applications operating at scale. Specifically, we study the skewed distribution in terms of operation type (read-heavy) [36] and data access (hot-spots / high contention settings). We study the effect of read-heavy workloads and high contention transactional settings in a distributed database architecture, by using CockroachDB [33]. Distributed database architectures provide support for both fault-tolerance and performance concerns via data distribution and replication. CockroachDB is a distributed database that employs Raft for consensus and builds a transaction layer on top of Raft, which uses a pessimistic timestamp ordering mechanism for concurrency-control.

To efficiently tackle read-heavy scenarios in a replicated environment, we design an approach to scale linearizable read operations by reading from a quorum, without going through the leader [36]. The technique ensures that a read operation still satisfies linearizability, and returns the latest value of a data item. We also design a scheme to combine the proposed **strongly-consistent quorum read** [36] scheme, with the existing leader-based reads, in a consensus protocol like Raft. Combining the two schemes in such a way leads to more uniform utilization of the cluster (both leader and non-leader nodes), and provides a mechanism to configure the system to trade-off between read and

write latencies. Extensive evaluation on AWS Cloud Platform demonstrates that the proposed approach results in higher throughput and improved read/write latencies with read-heavy workloads, and also better utilization of the non-leader nodes in failure-free scenarios.

In a distributed database setting like CockroachDB, high contention settings have a big impact on the transaction throughput and abort rate. CockroachDB employs pessimistic timestamp ordering for concurrency-control, to ensure serializable order of transactions. The pessimistic timestamp ordering leads to a high number of aborts under contention, due to conflict resolution based on static timestamps. To address performance concerns in the presence of hot-spots, we **integrate an existing dynamic timestamp based technique, MaaT [37]** in the concurrency-control mechanism in CockroachDB. In MaaT, commit timestamps are allocated at the end of transaction. The commit timestamp is dynamically allocated based on the items accessed by the transaction and conflicting operations on these items, rather than just statically allocating a timestamp, based on the clock or a shared counter. MaaT targets to avoid aborts in cases where transactions are aborted due to violating a pre-determined order because of a fixed timestamp ordering, rather than the actual conflicting order of access of items. Past techniques proposed to reduce aborts in lock-free concurrency control techniques have been implemented and evaluated in standalone prototypes. Integrating the dynamic timestamp ordering technique in a full-featured database system like CockroachDB gives an insight into the performance characteristics of the dynamic timestamp ordering like optimizations and their impact on improving performance in high-contention settings. To the best of our knowledge, this is the first time when dynamic timestamping technique has been evaluated in an open source, commercial database setting.

1.3.3 Tackling Physical Heterogeneity and Data Processing Variety of IoT Applications

The third part of the dissertation studies the variety challenges of IoT applications. We propose a **multi-representation architecture catered for IoT applications** [38]. To support IoT applications, we design an architecture that can benefit from the advantages of using different data processing engines (supported by diverse multiple representations), while removing the bottleneck of continuous data transfer. IoT applications have to support write-heavy workloads, with continuous ingestion of incoming data from multiple sensors. To handle these workloads efficiently, a unified strategy is used to update the multiple representations, rather than ingesting the data at a representation, and then employing continuous ETL or CDC pipelines to transfer data to other representations. A deterministic ordering scheme is proposed to update the different data representations. The deterministic approach is suited to IoT applications, which do not have to support general-purpose transactions. The approach guarantees that there is one global order enforced at all representations, and, also helps in optimizing for latency of new updates being reflected in queries, by removing the data transfer pipelines. The architecture also enables providing efficient support for specialized IoT analytics like pre-defined aggregates with low latency, by employing different schemas at representations and using the deterministic mechanism to incrementally update the aggregates.

IoT applications handle data from different physical sensors and integrating data from multiple diverse sensors provides the ability to get deeper insights into the data. The data management system should provide the IoT application developer with abstractions to perform such an integration. Currently, data management systems provide either transactional abstractions, which are not suited for IoT data, or simple key-value operations which burden the application developer with complexity of the integration.

We propose **Model-based Operators** as abstraction end-points to the IoT application developer. Traditional transactions imply that an event has taken place on the premise that all the sub-events comprising the transaction are executed atomically. For example, a transaction transferring money between two accounts is said to be executed successfully if both the accounts are updated atomically. Whereas, a successful model-based operator execution implies the execution of an event, along with the certainty in the execution, given that a pre-defined number of specified events is said to have occurred within a given number of sub-events (or time interval). For example, an abnormal vibration in a turbine is detected if a defined percentage of sensor values in a given period are over a pre-defined threshold. Abnormal vibration detection can be encapsulated as a model-based operator execution. The sensor value threshold, the time period and specified percentage for successful execution in this case, can be seen as a model specified by the application developer. The operator is successfully executed if the specified model is satisfied. We also propose **M-Stream**, a computation framework which builds on model-based operators to support continuous monitoring needs of IoT applications, while handling the underlying uncertainty of the sensor data. In M-Stream, model-based operators are combined in the form of a dataflow graph to support the application needs.

1.4 Research Contributions

This dissertation proposes solutions which tackle the diverse challenges and issues posed by the variety of data and heterogeneity of data infrastructure. The presented protocols and systems address varied aspects of variety: namely challenges from different data formats and structures, access methodologies, and heterogeneity within the workloads. The dissertation also explores the variety challenges of IoT applications, an emerging class of data-intensive applications, which have to deal with heterogeneity of

the sensors, as well as diverse data processing demands. Extensive evaluation studies are performed on public cloud platforms and private clusters, to demonstrate the effectiveness of various techniques. In particular, the dissertation makes the following contributions:

- We propose a hybrid scalable multi-representation cloud store, Janus, which supports storing data in multiple representations, and enables real-time analytical results, while supporting a high throughput of transactions. Janus employs a novel data pipeline, which supports real-time, consistent, continuous transfer of transactional data, in the presence of distributed transactions and diverse partitioning strategies at different representations.

- A multi-representation data-processing framework, Typhon, is proposed, which defines a novel formal consistency model catering to heterogeneous data. Typhon employs a notion of entities, which link the data present in diverse representations. Typhon defines implicit causal order at the granularity of the entities, to capture the logical order intended by the application user. The formal model is integrated with the existing conflict-graph model in databases, and can be used to reason about consistency guarantees provided by protocols in a setting where application data is spread across representations.

- Based on the consistency semantics defined in Typhon, we design a protocol, Cerberus, which provably enforces implicit causal order for operations accessing data across representations / processing engines. Cerberus adopts a single-entity transaction model, which enables providing transaction guarantees at the fine granularity of an entity, comprising related data items, without requiring a general-purpose transaction model. Cerberus is a lightweight and scalable protocol, designed using version vectors, and has a small overhead when compared to a solution providing no consistency guarantees for operations accessing data across representations.

- We also propose solutions to tackle the different aspects of workload distribution heterogeneity: read-heavy workloads and high data contention, in OLTP settings. These solutions are designed in context of a distributed database architecture, and are integrated in CockroachDB. The proposed solutions have been open-sourced and are available on Github. A strongly-consistent quorum read approach is designed, to support read-heavy workloads efficiently in Raft-like consensus protocols. Strongly consistent quorum reads can return the latest value without going through the leader node, and allow uniform utilization of the cluster in presence of read-heavy settings. A dynamic commit timestamp allocation technique is integrated within the concurrency-control mechanism of CockroachDB, to improve performance of OLTP workloads under high data contention.

- Data variety challenges of IoT applications are studied, and a multi-representation data management architecture catering to such applications, is proposed. A deterministic ordering mechanism is used to update all the representations. This results in removing the need for data-transfer pipelines, and enables efficient support of IoT application demands: updating the pre-defined aggregates with low latency and better support for write-heavy workloads.

- Model-based Operators are our proposed abstractions to enable application developers to integrate data from multiple diverse sensors, while capturing the underlying uncertainty of sensor data. We also propose a computation framework, M-Stream, to support continuous monitoring needs of IoT applications. M-stream uses model-based operators as building blocks, and combines them as a dataflow graph, to ensure that the uncertainty of the operator execution can flow through the processing pipeline.

1.5 Dissertation Organization

The rest of the dissertation is organized as follows. Part I proposes solutions to challenges arising out of variety in data structure and access methodologies. In Chapter 2, we present Janus, to efficiently handle both transactions and diverse analytical queries. Chapter 3 describes Typhon and Cerberus, which define consistency semantics for heterogeneous data.

In Part II, we present techniques to effectively handle distribution heterogeneity within the workloads, in a distributed OLTP database setting. Chapter 4 proposes a protocol to scale read-heavy workloads. In Chapter 5, we integrate and evaluate a dynamic timestamp allocation based concurrency-control scheme, for reducing aborts in high contention workloads.

Part III presents our work for handling the variety of IoT applications. Chapter 6 proposes a multi-representation based architecture catered for the IoT ecosystem. Chapter 7 introduces M-Stream, which enables continuous monitoring over uncertain sensor data, from a diverse set of sensors.

The dissertation concludes with a summary and future directions in Chapter 8.

Part I

Variety in Data Structure and Access Methodologies

Chapter 2

Janus: A Hybrid Scalable Multi-Representation Cloud Datastore

2.1 Overview

In this chapter, we focus on the challenge of supporting diverse access methodologies for applications, and enabling real-time analytics for applications operating at large-scale. Many emerging web-applications have to support both high frequencies of updates as well as diverse real-time analytics. Managing and analyzing advertising click streams, and retail applications employing Just-in-time inventory management are some scenarios where insights into data are needed in real-time. In all the examples mentioned, real-time analytics must be considered while supporting high ingestion rates of inserts, updates and deletes. Applications base their decisions on analytical operations, using insights from historical data as feedback into the system. The need for a tighter feedback loop between updates and analytics has created the demand for fast real time analytical processing [39].

Owing to the different characteristics of updates and analytics, many systems use different data representations to serve them. OLTP systems typically use row-based storage [20], systems targeting OLAP workloads might use column-based storage [16, 26], and graphs [17] might be used to store networked-data. Multi-engine architectures (1.2.1) store copies of data in different representations, to help in performing the update and analytics operations on the representations most suited for them. In such settings, one copy of the data can be designated as update-oriented and others as read-oriented.

A major challenge for data-intensive applications is handling large-scale data. Cloud datastores [12, 40] have been proposed to handle large-scale data and adopt scale-out techniques as a means to support data processing of such datasets. Data partitioning is a widely used technique for supporting scale-out, where different partitions are allocated to different servers. Hence, the data processing architectures supporting large-scale data need to support partitioning. Partitions are organized to limit the amount of cross-partition operations. But in cases where partitioning is not perfect, cross-partition operations have to be supported.

To support diverse workloads and enable real-time analytics for large-scale data, we propose and design a hybrid partitioned multi-representation cloud datastore, *Janus*. It maintains copies of data in different representations, and each of the representation supports partitioning. One representation supports transactional updates and the other representations are designated for analytics, and are read-only. Janus handles the execution of distributed transactions to ensure transactional consistency of operations over multiple partitions. To support different characteristics of diverse representations, Janus allows different partitioning strategies for the different representations.

Janus proposes a novel *data movement pipeline*, which ships data from each update-oriented partition in batches. These batches are then applied at the corresponding read-oriented partitions. Unlike existing hybrid representation storage systems [41, 42, 43] and

Change Data Capture (CDC) pipelines [27, 44], the data movement pipeline in Janus supports partitioning and handles both distributed transactions and different partitioning strategies. The capturing of changes as batches at update-oriented partitions is closely integrated with the concurrency-control mechanism and the distributed commit protocol. We devise a *graph-based dependency management* algorithm for applying batches from the update-oriented partitions, across to the read-oriented partitions. The end-to-end pipeline to move the data, across the partitioned representations, is developed to ensure no disruption in transactional execution while resulting in minimal delays in the incorporation of updates into the read-oriented partitions, and ensuring the consistent ingestion of transactional data. We now provide more details into Janus's design.

2.2 The JANUS Design

Janus enables the efficient execution of both transactional updates and *consistent* read-only analytical operations, at scale. Janus provides serializable isolation for transactional updates. Janus also provides the guarantee that any analytical query would not observe the partial affect of any transactional update and would observe the effect of the transactions in their serialization order. Next, we briefly discuss the major components of Janus. The system design is illustrated in the Figure 2.1.

Execution Engine: Application clients send requests to the execution engine which then determines whether the request corresponds to an update transaction or an analytical query. Transactional update requests are routed to the update-oriented representation and the read-only analytical queries are sent to the read-oriented representation, as shown in Figure 2.1. The execution engine also maintains the metadata pertaining to the location of data items, which is used to route queries to the appropriate partitions. The execution engine is a scalable middleware layer, similar to ones used in large-scale

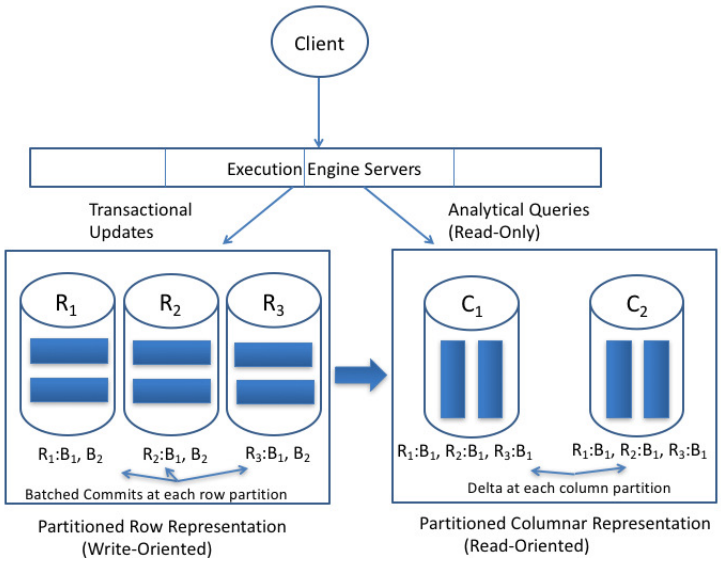


Figure 2.1: Janus Design

partitioned and replicated environments [40, 12]. Like in [12], the execution engine is scaled by replicating the partitioning information on multiple servers, and using these multiple execution engine servers to route operations to appropriate partitions.

Update-Oriented Representation: Janus stores a copy of the data in an update-oriented representation. The update-oriented representation supports single-partition as well as cross-partition transactions. In Figure 2.1, Janus uses row-wise storage as update-oriented and splits the data into three partitions, R_1, R_2 and R_3. When a transaction commits, the changes made by the transaction are stored in an in-memory append only structure, referred to as a *batch*. A batch stores changes made by committed transactions at an update-oriented partition and is used to commit these changes at the read-oriented representation. The methodology for creating consistent batches is described in the Section 2.4.1 and 2.4.3.

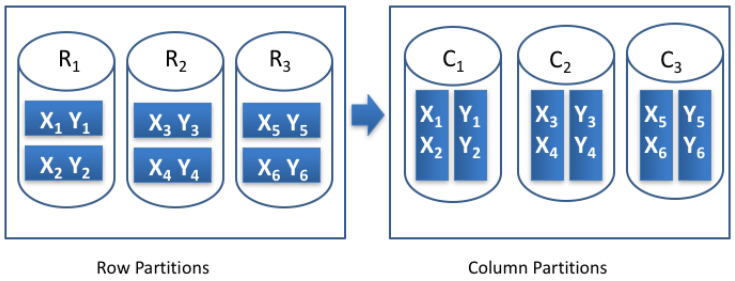
Read-Oriented Representations: Janus also stores data in other representations, which are designated as read-only. Every partition of a read-oriented representation consists of a persistent image and a *delta of changes*. The delta constitutes of incoming

batches from the update-oriented representation which are yet to be ingested into the partition. In Figure 2.1, Janus uses column storage as read-oriented and splits a copy of the data into two partitions, C_1 and C_2. In the Figure, the row partitions and the column partitions use different partitioning strategies, and the number of row partitions (3) is different from the number of column partitions (2). Three batches arriving at C_1 are represented by R_1:B_1, R_2:B_1 and R_3:B_1. A read-oriented partition receives batches from all the update-oriented partitions that have changes mapping to that particular read-oriented partition. Updates are made to a read-oriented partition by applying the batches to the persistent image of that partition. A graph dependency management algorithm is used for applying the batches, ensuring that the consistency of data is preserved (Section 2.5.1).

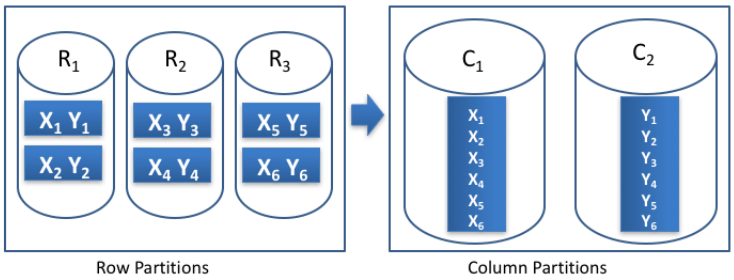
2.3 Hybrid Partitioned Row and Column Data Management

In this dissertation, we focus on designing an instance of Janus to support OLTP and OLAP workloads. Since, row and column representations have been widely used for OLTP and OLAP workloads [43, 24], we choose these representations as update-oriented and read-oriented representations respectively. First, we discuss the partitioning of data at both the representations. Then, we explain transaction processing (Section 2.4) and the ingestion of the changes at the read-oriented partitions (Section 2.5). Although the techniques are presented for row and column, they can be applied to different representations, chosen based on the workloads being handled.

Partitioning: Janus allows applications to partition their data. Both the row and the corresponding column data can be partitioned. The row-oriented data is divided



(a) Partitioning - Case 1



(b) Partitioning - Case 2

Figure 2.2: Diverse Partitioning Strategies

into n partitions, R_1, R_2 ... R_n and the column data is partitioned into m partitions, C_1, C_2 ... C_m. The partitioning strategy of the columns may or may not correspond to that of the rows. Janus provides this flexibility because different applications have varied characteristics governing the analytical workloads. For example, a multi-tenant data platform will partition the data across the tenant boundaries. Hence, storing the column version of each partition is suitable, as there would not be a need for analytical operations on columns across different partitions. Whereas an e-commerce store might store customers in various regions in different row partitions. In such cases, there might be a requirement to perform analytics on columns across the row partitions to collect aggregate statistics across regions. Therefore, a partitioning strategy which stores some of the columns entirely in a single column partition (vertical partitioning) may be more efficient. Two different instances of partition mapping schemes that can be employed are:

- The column partitions can be partitioned corresponding to the row partitions.

There would be n column partitions, $C_1 \ldots C_i \ldots C_n$, where C_i is the columnar version of the row partition, R_i, as represented in Figure 2.2a.

- Store the entire column together in a separate partition. This would lead to r column partitions, where r is the number of attributes in the partitioned table. Each column partition, C_i would have a column stored in its entirety, containing data from across row partitions, as shown in Figure 2.2b.

2.4 Transactional Processing

Janus handles both OLTP and OLAP workloads. OLTP operations are executed on the row partitions and column partitions are continuously updated by bulk committing the results of the updates as batches. The read-only analytical queries are then executed at column partitions.

When a transaction T arrives at the execution engine, it looks up the metadata to determine the partitions involved in the transaction. The transaction is then sent to the corresponding partitions. Each row partition employs a local concurrency control mechanism to manage the concurrent execution of transactions. In this specific implementation of Janus, each row partition uses strong strict two-phase locking (2PL) as the concurrency control method [45]. However, Janus can be adapted to other *commit-order preserving* serializable concurrency control mechanisms. The concurrency control protocol needs to be commit-order preserving to ensure the correctness of the batch generation scheme. In addition, distributed transactions employ two-phase commit (2PC) for atomic commitment across partitions. When a transaction commits, the changes made by the transaction at a partition are stored in a *batch*. We first describe the scheme for creating a batch for single-partition transactions. Then, we discuss the support for distributed transactions in Section 2.4.3 and describe the mechanism for sending the batches to the

column partitions in Section 2.4.4.

2.4.1 Consistent Batch Generation for Single Partition Transactions

Transactional changes are stored in *batches* at every row partition. In this subsection, we consider transactions that are restricted to a single-row partition. A batch is an in-memory append only structure and comprises a set of transactional changes executed at a row partition. The batch structure is analogous to the partial transactional redo command log used for recovery by main memory databases. A tag is associated with each batch, indicating whether the batch is in an *active* or *closed* state. Each row partition has only one active batch at any instant. Once an active batch is closed, it is sent to the column-oriented representation to ingest the changes present in the batch. Each batch is also associated with a *version number* which determines the order of the batch, as compared to other batches from the same partition. The batches are continuously shipped and applied to the corresponding column partitions to keep them updated. The size of the batch shipped is a trade-off between maximizing the data freshness of analytical results and restricting the overhead involved in generating, shipping and applying the batches. To manage the size of the batch, batches are shipped after a pre-defined period. This period is named *batch shipping frequency*. The batch generation scheme ensures that all transactions executed at the row partitions are captured and the serialization order of such transactions is maintained. Each batch is also assigned a unique id, referred to as *batch-id*. The batch-id is a combination of the row partition and the batch version number. A batch with batch-id R_i:B_j refers to changes included in batch with version number B_j from row partition R_i. In Figure 2.1, R_1:B_1 is a closed batch and has been shipped to the column representation. Whereas, R_1:B_2 is an active batch and any

transactional changes done at R_1 will be appended to $R_1{:}B_2$.

On a commit, all the write operations of the transaction are atomically appended to the batch. All the changes at a particular row partition resulting from a single transaction are stored in the same batch. After a pre-set threshold, the batch is *marked closed* and a new batch is created and *marked active*. Transactions which commit after the old batch is closed are added to the new batch. The switch from the old batch to the new batch is done atomically. The new batch is assigned a version number, which is one more than the version number of the closed batch. Once a batch is closed it is shipped to the column partitions. Closed batches at a row partition can be discarded once the corresponding column partitions acknowledge its receipt.

Batch generation is integrated with the existing transactional log mechanism for handling failures. When a batch is closed, an additional record is written to the transactional log, noting the switch to the new batch. During the recovery phase, the batch generation scheme recovers the state of the last active batch before the crash. In addition to the existing recovery operations, the record marking the closure of the last batch is noted and is used to reconstruct the state of the active batch before the crash. Transactions with a commit record after the record marking the closed batch are added to the current active batch.

2.4.2 Batch Properties

The batching scheme ensures several invariant properties, needed for generating consistent batches. We now list the properties and argue that the batching scheme provides these guarantees.

Property 1 *A closed batch only contains operations from committed transactions and each transaction is included in its entirety.*

As each transaction is added to an active batch once it is committed, this ensures that only committed transactions are present in a batch. When a transaction is committed, all its changes are added to a batch. Thus, transactional changes are present in their entirety for every transaction. This property ensures that all transactional updates performed at the row partitions are included in the batches.

Property 2 *For any two transactions T_1 and T_2 present in batch B_i, if T_1 is serialized before T_2, $T_1 \rightarrow T_2$, then T_1 is present before T_2 in B_i.*

Transactions are appended to a batch at commit. So, the order in which transactions are appended to a batch follows the commit order of transactions. This case is true if the concurrency control protocol is commit-order preserving serializable. As we are using strong strict 2PL for concurrency control, this condition is true in Janus. Hence, if $T_1 \rightarrow T_2$, then T_1 would have been appended to B_i before T_2, and as a batch is append-only, T_1 will be present before T_2 in B_i. This invariant establishes that the serialization order is preserved while appending transactions to a batch.

Property 3 *For any two transactions T_1 and T_2, where T_1 is serialized before T_2, $T_1 \rightarrow T_2$, then if $T_2 \in B_i$, where B_i is a closed or an active batch, then either $T_1 \in B_i$ too or $T_1 \in B_j$, where B_j is closed and $V(B_j) < V(B_i)$, where $V(B_i)$ is the version number of batch B_i.*

The order in which transactions are appended to batches follows the commit order of transactions. As T_1 is serialized before T_2, it either would have been applied to the same batch or an earlier batch. As batch version numbers increase over time, if T_1 was applied to an earlier batch, the corresponding batch would have a version number less than version number of batch B_i. This condition guarantees that batch version order corresponds to serialization order of transactions.

Property 4 *If a transaction T_i belongs to a batch B_j before a failure, then after recovery from failure, T_i will still be present in batch B_j.*

When a transaction commits, it is written to an active batch. If a failure is encountered, the active batch is reconstructed from the write-ahead log during recovery, by appending the update records of every transaction with a commit record after the record indicating the closure of the old batch. Since changes corresponding to any committed transaction T_i are present in the write-ahead log, any transaction T_i which belongs to the active batch B_j before the failure, will also belong to the active B_j after recovery. All the batches which have been closed have either been already shipped or they are recovered using the same protocol, hence preserving the invariant that each transaction is in the same batch as it was before the failure. This property ensures that batches are correctly recovered after failure.

These properties guarantee that the batching scheme generates consistent batches comprising all transactional changes at a row partition, even in the presence of failures.

2.4.3 Distributed Transaction Support

To enable operations accessing data across partitions, Janus provides the ability to perform distributed transactions. Janus employs the two-phase commit (2PC) protocol for executing distributed transactions. The execution engine acts as a coordinator of 2PC. When the transaction commits, updates of the transaction corresponding to a row partition, are stored in the batch at that partition. Hence, transactional changes of a distributed transaction may be present in multiple batches across different partitions. Furthermore, since the partitioning scheme supported at the row and column representation might be different, changes at a single column partition might correspond to batches from multiple row partitions. Consider a distributed transaction, dt, which changes a col-

umn attribute A in tuple x at R_1 and y at R_2 in Figure 2.2b. The column representation uses the partitioning strategy illustrated in the figure, leading to entire column A being stored in column partition C_1. Suppose the changes done by dt are present in R_1:B_1 and R_2:B_2. Since, the changes of dt are present across different batches, the effect of these batches should be atomically visible. Hence, the algorithm for creating the batches and applying them to the column partitions needs to be carefully designed to guarantee the consistency of analytical results.

We need a method to identify the batch dependencies at the column partitions. For capturing these dependencies, *metadata* is added to each batch, which provides information about the distributed transactions present in the particular batch. The metadata includes the batch-ids of the set of batches from different row partitions, involved in distributed transactions present in the given batch. Janus integrates the bookkeeping of the metadata with the two-phase commit protocol. The needed metadata is piggybacked during the various phases of two-phase commit of a distributed transaction.

- **Prepare Phase.** During the prepare phase of two-phase commit (2PC) of any distributed transaction, each participant row partition *piggybacks* the information about the batch version number to the response of the prepare message.

- **Commit Phase.** Subsequently, if the transaction is committed, the 2PC coordinator piggybacks the batch-ids of all the batches having changes pertaining to the distributed transaction to *each* row partition, along with commit status information.

When a distributed transaction is added to a batch, the set of batches with changes pertaining to the transaction (sent by the 2PC coordinator along with the commit status), are added to the metadata of the batch. Each row partition ensures that the current active batch is not closed between sending the batch-id to the 2PC coordinator and the

addition of such a transaction to the corresponding batch. In the example introduced earlier, updates and inserts corresponding to the distributed transaction *dt* are present in batches, R_1:B_1 and R_2:B_2. Then, R_2:B_2 is added to the metadata of batch R_1:B_1 and vice-verse. This added metadata is used to ensure that the data in column partitions remains consistent, as we describe in Section 2.5.1.

2.4.4 Batch Shipping

After a batch is closed, it is shipped to the column partitions. Batches from each row partition are sent to *all* the corresponding column partitions. This is depicted by R_1:B_1, R_2:B_1, R_3:B_1 at column partition C_1 in Figure 2.1. Each row partition contacts the execution engine to retrieve the metadata pertaining to the column partitions corresponding to the row partition, based on the partitioning strategies employed. This metadata is cached at the partitions. The batches can be *pre-filtered* by dividing them into sub-batches corresponding to the different column partitions before shipping the batches. Alternatively, in an approach we name as *post-filtering*, the *entire* batch is sent to all the corresponding column partitions. Each column partition only applies the changes which correspond to that partition.

Batch Filtering. The decision of whether to post-filter or pre-filter can be based on a number of factors. The overall filtering and batch shipping cost can be divided into two parts: computation and data transfer. The computation cost is a function of cpus at the row and column partitions. In pre-filtering, the computation cost of filtering falls upon row partitions. In post-filtering, the computation cost is divided among the column partitions. The data transfer cost is a function of the available network bandwidth between row and column partition servers. As the average number of column partitions mapped from a row partition increase, the gap between the bandwidth consumed during

post-filtering and that consumed in pre-filtering, increases. One the other hand, the post-filtering approach has the advantage of offloading the filtering of batches from row partitions. As Janus aims to minimize any affect on transactional throughput, Janus employs post-filtering.

Batching Threshold. A batch is closed after a fixed duration known as batch shipping frequency. The shipping frequency provides a time-based threshold to restrict the size of the batch and to ensure that batches are regularly shipped and ingested at the column partitions. Although batch shipping frequency provides a simple threshold, which can be easily adjusted, it has some drawbacks. Since, a shipping frequency only provides a time-based mechanism, it can lead to uneven batch sizes. If long running transactions with a large number of updates are present in the workload or the workload is write-heavy, this can lead to an increase in the batch size (in terms of the number of updates), which can increase the time to ingest batches at the column partitions, and thus resulting in a larger delay for an update to be reflected at the columnar representation. To avoid this bottleneck, we add the ability to combine a time-based threshold with a threshold on the number of updates in a batch (referred to as *batch shipping update-threshold*). If the number of updates in the batch goes above a certain threshold, then the batch can be closed and shipped to the column partition without waiting for the batch shipping frequency duration.

2.5 Supporting Analytical Processing

Janus executes read-only analytical queries on the column representation. In Section 2.4, we discussed the protocol for creating batches containing transactional changes occurring at row partitions. These batches are then sent to column representation. Each column partition consists of a persistent column-oriented copy and a delta as shown in

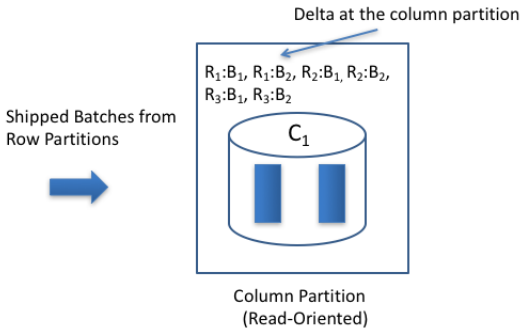


Figure 2.3: A Column Partition

Figure 2.3. The delta consists of incoming batches from different row partitions mapping to the particular column partition. We describe the protocol for merging the incoming batches to the column partition to guarantee that a read-only query accessing any *single* column partition will be consistent, even in the presence of distributed transactions at the row partitions. We then provide an extension to guarantee the consistency of analytical queries spanning *multiple* column partitions. Janus ensures that any analytical query:

- Observes the serialization order of transactions executed on the row-oriented representation
- Does not observe the partial effect of any transaction

As analytical queries can be long running, they are executed on a consistent snapshot of the data.

2.5.1 Consistent Single Column Partition Analytical Operations

If an incoming batch from a row partition does not have any change corresponding to a distributed transaction, then it can be applied atomically to a column partition.

This ensures that analytical operations will observe the effect of each transaction present in the batch in its entirety. The update operations in an incoming batch are grouped as a single transaction. The transaction is then executed using the concurrency control scheme at the column partition. Applying the batch as a single transaction ensures that all the changes in the batch are observed atomically. Batches from each row partition are ingested in order of their version numbers. As Janus ensures that each analytical query observes a consistent snapshot of the data, any index on the column partition will be updated synchronously with the ingestion of the batch. This mechanism ensures that the serialization order of transactions executed at a single row partition is maintained at the column partition.

If distributed transactions are present in the workload and the row and column partitions are not aligned, then the changes from a distributed transaction can arrive in batches from different row partitions, as described in Section 2.4.3. This may lead to analytical operations not being consistent. Continuing with the example from Section 2.4.3, the changes done by dt are present in R_1:B_1 and R_2:B_2. R_1:B_1 includes changes to column A at tuple x, whereas R_2:B_2 includes changes to column A at tuple y. Consider the scenario where we atomically apply R_1:B_1 to the column partition and then execute an aggregate query on column A involving tuple x and y, before applying R_2:B_2. The result of such a query will be inconsistent as it would include partial changes from transaction dt. Hence, batches with partial changes from distributed transactions must be ingested atomically.

To ensure the consistency of analytical operations in the presence of distributed transactions and different partitioning strategies, changes are ingested to the column partitions by a *graph-based dependency management algorithm*. The presence of distributed transactions leads to dependencies across batches from different row partitions. Such batch dependencies are included in the metadata of each batch (Section 2.4.3). At each column

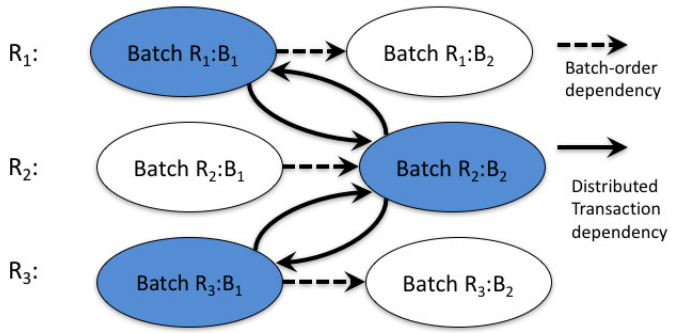


Figure 2.4: Batch Dependency Graph

partition, the batch dependencies are modeled as a directed graph, referred to as the *batch dependency graph*. As described earlier, each batch has a unique id. The id of the batch is a combination of the row partition id and the batch version number, which guarantees its uniqueness. An incoming batch, with an id α, is represented as a node with outgoing and incoming edges. An incoming edge refers to a batch upon which batch α depends on. Whereas, outgoing edges refer to the other batches dependent on batch α. The dependencies between the batches can be classified in two categories. Apart from the dependencies introduced by distributed transactions, referred to as *distributed transaction dependency*, each batch from a particular row partition is also dependent on the previous batch from the same row partition. This dependency is referred to as *batch-order dependency* and is added to capture the condition that batches from any particular row partition are ingested in their batch-id order. An example of a batch dependency graph at a particular column partition is shown in Figure 2.4. In the example shown in the figure, batch with id R_1:B_1 is dependent on batch with id R_2:B_2 and vice-verse. This is a dependency arising out of a distributed transaction, i.e. a distributed transaction dependency. Batch R_1:B_2 is dependent on R_1:B_1 but not vice-verse, which represents a batch-order dependency. This dependency ensures that batch R_1:B_2 can not be ingested before the batch R_1:B_1.

Any batch can only be applied to the column partition when all its dependencies are satisfied. Batch dependencies are represented by the incoming edges of the node. This condition implies that a batch can only be ingested either after all the batches it depends on are ingested or it can be ingested atomically with dependent batches. Given this condition, finding a set of batches to apply at the column partitions is equivalent to finding **strongly connected components** (SCC) in the batch dependency graph. Such a SCC should not have any incoming edge connected to any node outside the SCC. This ensures that any batch does not have any dependency apart from the batches in the SCC. Batches in such a SCC are then atomically ingested into the column partition. All the nodes of the strongly connected component can then be deleted from the batch dependency graph. In the Figure 2.4, batches $R_1{:}B_1$, $R_2{:}B_2$, $R_3{:}B_1$ form a strongly connected component, which will be ingested atomically after the ingestion of batch $R_2{:}B_1$. We implement the detection of SCC in the batch dependency graph using the union-find data structure [46].

The batch ingestion scheme also handles crash failures of column partitions. On ingestion of a SCC, the column partition also persists the batch-ids ingested corresponding to the row partitions it maps too. As noted in Section 2.4.1, a row partition only deletes a batch once it receives an acknowledgment from the corresponding column partition(s). The acknowledgement is only sent after the batch has been ingested at the column partition. On recovery from a crash, a column partition informs the corresponding row partitions of the last batch-ids ingested. The row partitions will then start sending the batches ordered after the last batch-id ingested. As the batch dependency graph at the column partition comprises the active batches, it will be reconstructed when the row partitions start sending the batches which have not been ingested at the column partition. Since, we guarantee that only ingested batches are acknowledged, row partitions will have all the batches which have not been ingested. Hence, even on crash failures, no

batches will be missed and the batches will be ingested in order.

2.5.2 Supporting Multi Column Partition Analytical Operations

The graph-based dependency management algorithm ensures that any analytical query accessing a single column partition will return consistent results. We now provide an extension to support multi column partition read-only queries. A query which accesses data at multiple column partitions, might access an inconsistent snapshot if a batch from a row partition has only been ingested at some of the column partitions. Consider that a transaction, t, updates columns A in tuple x and column B in tuple y at R_1. Suppose, the column representation uses the partitioning strategy shown in Figure 2.2b, where column A is stored in column partition C_1 for both x and y and column B is stored in C_2. Updates from transaction t are added to $R_1{:}B_1$. Now, consider a query which calculates an aggregate on column A with a select condition on column B. If the batch $R_1{:}B_1$ is only ingested at one of the column partitions, C_1, then the result of the query would be inconsistent. An inconsistency could also occur even if partitions are aligned but distributed transactions are present. Consider the distributed transaction dt, from Section 2.4.3. The changes done by dt are present in $R_1{:}B_1$ and $R_2{:}B_2$. $R_1{:}B_1$ includes changes to column A at tuple x, whereas $R_2{:}B_2$ includes changes to column A at tuple y. Now, suppose that column partitions use the partitioning strategy in Figure 2.2a. Then $R_1{:}B_1$ would be sent to C_1 and $R_2{:}B_2$ would be sent to C_2. Any query executing at both C_1 and C_2 should atomically observe the changes in these batches.

We design a multiversioning scheme and combine it with the graph-based dependency management to ensure the efficient execution of consistent multi column partition analytical operations. Each column partition updates its state by ingesting changes from

the row partitions, using the graph dependency algorithm introduced in Section 2.5.1. Each ingestion leads to the creation of a new version. A column partition can be represented in terms of row partitions, that map to the column partition. A column partition is represented as a *version vector* of batch versions from these row partitions. Suppose column partition C_1 maps to row partitions, R_1 and R_2 and ingests batches R_1:B_3 and R_2:B_2 to create a new version. Then, the new version at C_1 can be represented as the version vector: (B_3, B_2). Each version of the data at the column partition has a vector, known as *version vector tag* associated to it. This version vector tag is used to compare the recency of the versions. The entries of the version vector tag of the ingested version comprise the batch version numbers corresponding to the row partitions that map to the column partition. Among two versions of data at a column partition, the version with a higher version vector tag is the newer version. The version vector tag comparison is based on the corresponding batch version numbers, which comprise the entries of the version vector tag. The information about the creation of each version is sent to the execution engine.

For each multi-column partition query, the execution engine determines the latest compatible version at each column partition involved in the query. When a multi-column partition query arrives, the execution engine first determines the column partitions involved in the query. The execution engine then determines the version with the most recent version vector tag, which corresponds to the latest ingested batch versions of all the corresponding row partitions. If some of the column partitions involved in the query receive data from a common subset of row partitions, then for each such row partition, we include the latest common batch version ingested at the column partitions. For example, suppose the query involves column partition C_1 and C_2. C_1 maps to R_1 and R_2 and C_2 maps to R_2. We determine the most recent batch version of R_1 at C_1, and the most recent version corresponding to R_2, present at both C_1 and C_2. Then, using

the batch dependency metadata defined in Section 2.4.3, the latest version vector tag is determined, with the condition that the batch version entry (in the latest version vector tag) for each mapping row partition, satisfies all the batch dependencies. Suppose (B_3, B_2) is the most recent version vector tag corresponding to the column partitions C_1 and C_2. The batch versions are then iterated to find a version vector tag equal to or older than the tag, (B_3, B_2), which satisfies the dependencies of all batches version entries involved. Now suppose, $R_1{:}B_3$ was dependent on $R_2{:}B_3$, then the dependencies would be checked with the version vector tag (B_2, B_2). This process continues until a version vector tag is found that satisfies all the dependencies. At each column partition involved in the multi-column partition query, the query is then executed on the version corresponding to the found version vector tag. By ensuring that the batch dependencies are satisfied across column partitions, Janus guarantees that any multi-column partition query accesses a consistent version of the data.

Since this scheme can result in many versions, there is a background process which garbage collects older versions. The background job is also triggered by the execution engine, since it tracks the batch dependencies and versions maintained at each column partition. The execution engine maintains the global version of the batch dependency graph. When a Strongly Connected Component (SCC) in the graph is ingested at all the corresponding column partitions, then the versions corresponding to older batches can be garbage collected at all those column partitions. The ingestion of a SCC of the batch dependency graph signifies that a newer version (with a higher version vector tag) is available at all the dependent column partitions and all the dependencies of the newer version are satisfied. This is precisely the condition for using a particular version for an analytical query. The presence of such a version at the column partition implies that query would never be scheduled on a older version. Hence, older versions at the column partitions can be garbage collected. Suppose $R_1{:}B_2$ and $R_2{:}B_2$ form a SCC in the batch

dependency graph. When batches R_1:B_2 and R_2:B_2 are ingested at all corresponding column partitions, then any version at a column partition with a version vector tag smaller than (B_2, B_2), can be garbage collected.

As the batch ingestion is handled independently by each column partition, no special recovery needs to be employed for multi-column partition queries. Each column partition recovers independently from crash failures. The multiversioning scheme determining the compatible version for a query only checks the ingested versions, and is independent of the recovery and ingestion mechanisms.

2.6 Evaluation

Janus is evaluated using a transactional YCSB benchmark. We focus on measuring data staleness at column partitions and evaluating the impact of batching scheme on transactional throughput. First, we briefly describe the benchmark and then give information about the experimental setup, baseline and metrics collected during the experiments.

2.6.1 Benchmark description

We design a benchmark, which is an extension of T-YCSB (Transactional YCSB) [47]. Apart from adding the ability to invoke transactions, the benchmark is adapted to a partitioning environment and can invoke a specified percentage of distributed transactions. The Yahoo Cloud Serving Benchmark (YCSB) [48] is a benchmark for evaluating different cloud datastores. YCSB sends single key-value read and write operations to the datastore. The workloads generated by the benchmark sends multiple such operations combined as transactions. Each transaction consists of a begin operation followed by multiple read and write operations, followed by a request to commit the transaction.

Each operation of a transaction is invoked as a blocking operation, i.e, the client thread blocks until it receives a response. Unless otherwise mentioned, each transaction constitutes 4 reads and a write. The percentage of writes is varied in some experiments. The benchmark also provides the ability to configure the number of clients spawned in parallel.

Workloads generated by the benchmark either comprise only single-partition transactions or a specified percentage of distributed transactions. Each transaction first picks a primary partition for the transaction. If there are no distributed transactions, then all read and write operations are uniformly distributed over the primary partition. When distributed transactions are present, each operation of a transaction has a uniform probability of accessing a partition other than the primary partition. If 10% of the transactions in the workload are distributed, then each operation has a 2% chance (since each transaction has 5 operations) of accessing a partition other than the primary partition.

To measure the performance of read-only analytical queries in Janus, the benchmark issues aggregate queries which calculate average, minimum and maximum values of an attribute.

2.6.2 Experimental Setup

The dataset is range partitioned over row partitions. Each row partition consists of 100,000 data items. Column partitions employ a different range partitioning scheme with each column partition comprising 200,000 data items. Hence two row partitions correspond to a column partition. This partitioning scheme is a combination of the partitioning schemes described in Figures 2.2a and 2.2b respectively. Each experiment uses this partitioning scheme. Each item in the dataset comprises of 2 attributes: a primary key and a value. Our standard deployment consists of 20 row and 10 column

partitions, along with 4 execution engine servers. As each of the row partitions, column partitions and execution engine servers is placed on a separate machine, the standard deployment employs 34 machines in total.

The evaluations were performed on AWS [49] (Amazon Web Service) EC2 (Elastic Compute Cloud). We employ m3.xlarge instances which have 4 virtual CPU cores and 15 GB memory. All machines were spawned in US East Virginia region in the same availability zone. MySQL [23] is used as the row-oriented storage engine and MonetDB [14] as the column-oriented storage engine. Note that Janus implements strict two-phase locking (2PL) concurrency control protocol, two-phase commit (2PC), as well as the batch generation and ingestion scheme. As the batch generation scheme is closely integrated with the concurrency control and the distributed commit protocol, we choose to implement both 2PL as well as 2PC at the row partitions. The implementation, therefore, does not rely on the concurrency control protocol of MySQL, but only employs it as row-oriented storage engine. On the other hand, each column partition uses MonetDB for storage as well as concurrency control. Hence, Janus employs MonetDB's optimistic concurrency control (OCC) mechanism. The concurrency control mechanism at each column partition is needed by the batch ingestion scheme to ensure the atomicity of batch ingestion. Row and column partitions reside on different EC2 machines. However, Janus can also be deployed with multiple row and column partitions being placed on the same machine. Janus is implemented in Java. It uses protocol buffers [50] for serialization and protobuf-rpc library for sending messages between the servers.

A number of application clients are spawned in parallel. Clients are co-located and uniformly distributed over the execution engine servers. Each client executes 500 transactions.

To analyze Janus's performance, we compare it against a baseline setup, where both transactions and analytics are performed on the transactional engine. To enable this

setup, we turn off the batching in Janus. This comparison enables us to assess if the batching scheme affects the transactional throughput. Both setups utilize strong strict 2PL as the concurrency control protocol, ensuring an equivalent comparison.

Unless otherwise stated, the batch shipping frequency in Janus is set to 250 ms. Batch shipping frequency is the time period after which a batch is closed and shipped to the column partitions. To ensure that all row partitions do not send batches to a column partition at the same instance, we add some random noise (+/- 20%) to the batch shipping frequency at every row partition. For a shipping frequency of 250 ms, each row partition would have a frequency in the interval of [200,300] ms. Later, we also evaluate the impact of batch shipping frequency and analyze the optimal shipping frequency for our standard deployment. We also perform an experiment where an update-based threshold is introduced for batch rollover, in addition to the batch shipping frequency. Measurements were averaged over 3 readings for smoothing any experimental variations. The measurements reported are described below.

Transactional Throughput This metric reports the number of transactions executed per second (tps).

Average Delay gives the measurement of data freshness in Janus. A delay value for a transaction includes the time period between the commitment at the row partition and its ingestion at the column partition(s). Hence, average delay includes the time period between adding the committed transaction to the batch and shipping the batch, and the time taken to ship the batch and the time for merging the batch at the column partitions. As batching is a feature of Janus and is turned off for the baseline evaluation, we only report this metric for Janus. Values reported are average of the mean delay values observed at the column partitions. The average delay metric gives a measure of freshness of the results returned by analytical queries. As row and column partitions are placed on different machines, average delay is measured across different machines. The

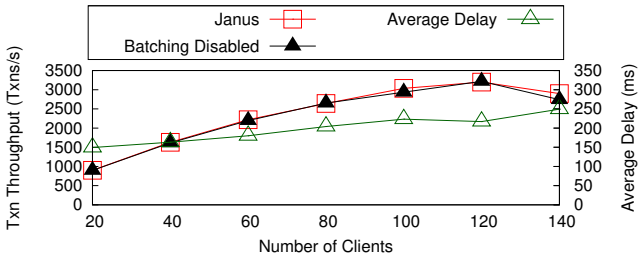


Figure 2.5: Scaling Up Clients

average delay measurement faced challenges involved in measuring time in a distributed system. Initially, high clock drift values were observed at the machines. To circumvent clock drift, networking time protocol (NTP) synchronization [51] was employed. The NTP utility in unix synchronizes the clock of the server with centralized time servers.

Query Response Time is reported for read-only analytical queries performed during the experiments.

2.6.3 Experimental Results

Scaling Up

Figure 2.5 evaluates the end-to-end performance of the data movement pipeline in Janus. We employ a standard deployment of 20 row and 10 column partitions. The number of clients running concurrently were varied from 20 to 140. In this set of experiments, the workload only consists of single partition transactions. In Janus, the transactional throughput increases from around 900 tps to 3197 tps as the number of clients increase from 20 to 120. On increasing the clients to 140, the throughput decreases slightly. The baseline scenario with batching disabled also achieves similar transaction throughput illustrating that Janus's batching scheme does not adversely affect transactional performance. Less than .1% of transactions were aborted for each case. As the highest throughput is observed with 120 clients, we employ 120 clients for later experiments.

Average delay was measured for Janus. Low average delay values were observed, ranging from 149 ms with 20 clients to 217 ms with 120 clients. The delay value increases with higher transactional throughput. As each batch comprises more transactions, it takes longer to ingest a batch into column partitions. The variance observed in delay values at different column partitions was 26.5 with 120 clients. This is low compared to the mean value of 217 ms. Figure 2.6 shows the observed delay at a column partition through the course of the experiment with 120 clients. The delay initially increases as batches start committing at the column partition and then remains constant as the system reaches a steady state.

These results demonstrate that Janus updates the column partitions in near real-time without impacting the transactional throughput.

Hot Spots. We also performed an experiment using a skewed workload distribution. In this experiment, 70% of the transactions access 30% of the partitions. The standard deployment of 20 row and 10 column partitions is employed and clients are scaled-up from 20 to 120. Batching still does not impact throughput and average delay was sub 750 ms, as seen in Figure 2.7. Delay values were higher and had greater variance due to the hot spot, as compared to results in Figure 2.5.

Varying Read:Write Ratio. Next, we vary the ratio of read to writes in the workload, as illustrated in Figure 4.3. Instead of issuing a single write for each transaction (20% writes), the overall percentage of writes in the workload is varied from 5% to 40%. The throughout slightly decreases as the percentage of writes in the workload increases. Average delay increases with the increase in percentage of writes, since each batch has more operations to ingest at column partitions. After the percentage of writes increases above 25%, delay increases at a high rate. This is because the batch shipping duration is not enough to cope with the time to ingest the batch at MonetDB, and the batches start getting queued at the column partitions. This illustrates that batch shipping frequency

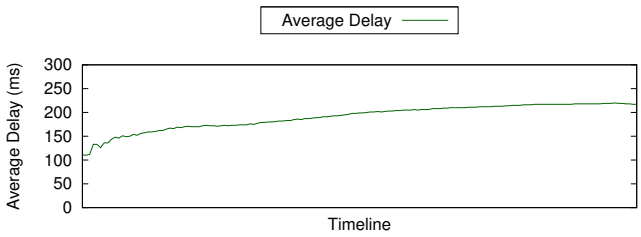


Figure 2.6: Average Delay Timeline

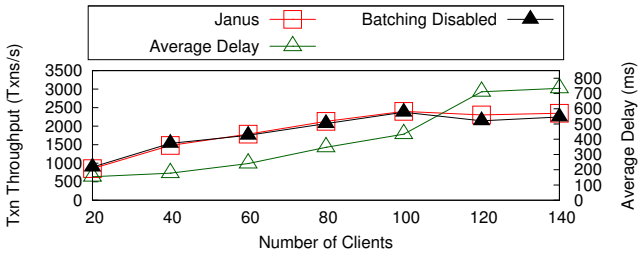


Figure 2.7: Scaling Up Clients with Hot Spots

needs to be carefully chosen based on the workload.

Impact of Distributed Transactions

Next, Janus is evaluated with workloads comprising distributed transactions. The standard deployment of 34 machines is employed, with the number of clients set to 120. The results are presented in Figure 2.9. The presence of distributed transactions results in a slight decrease in throughput as Janus has to employ two-phase commit for such transactions. The results also re-illustrate that the batching scheme does not affect transactional throughput. An increase in average delay is also observed, as distributed transactions lead to dependencies among the batches. The graph-based dependency management algorithm only ingests a batch either after all the dependent batches have been ingested or with dependent batches. After all the dependencies of a batch arrive at the column partition, the batch will be a part of one of the strongly connected components (SCC) detected by the ingestion scheme, and will be merged into the column parti-

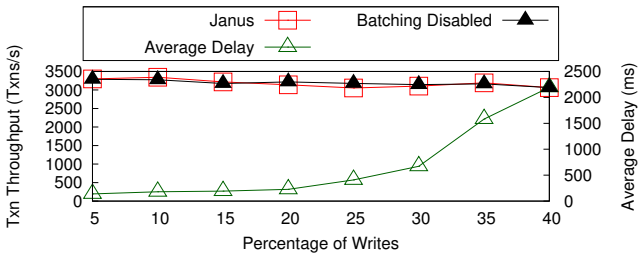


Figure 2.8: Varying Read:Write Ratio

tion. The experiments illustrate that distributed transactions have impact on the data freshness. However, even in the presence of distributed transactions, Janus is capable of updating the column partitions in 304 ms and 409 ms with 5% and 10% distributed transactions respectively.

Instrumenting Average Delay. We also instrument and sub-divide the average delay duration into 4 components: average waiting time for a transaction before a batch is shipped (row-wait), network transmission and processing time (network cost), waiting time at the analytical partitions before the entire SCC has arrived (column-wait) and the actual time to ingest the batch at MonetDB (batch-ingest). For 0% distributed transactions, the row-wait duration accounts for 67% of the time and the batch-ingest duration accounts for 30% of the average delay period. The column-wait time is negligible in this case (around 1%). As the distributed transactions increase to 5%, the column-wait duration goes up-to around 5% and batch-ingest duration goes up-to 32% (batch size increases because multiple batches are ingested together to satisfy batch dependencies). With 10% distributed transactions, the dependencies among the batches cause the column-wait duration to increase to 10% of the delay duration, and the batch-ingest duration fraction goes up-to 36%. The network cost was around 1% in all the cases. The sub-division of the average-delay illustrates how the dependencies among the batches due to distributed transactions lead to the increase of average delay period.

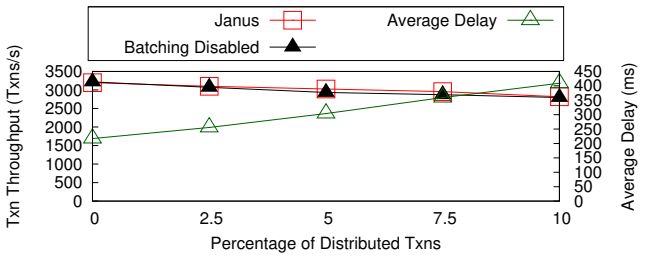


Figure 2.9: Impact of Distributed Transactions

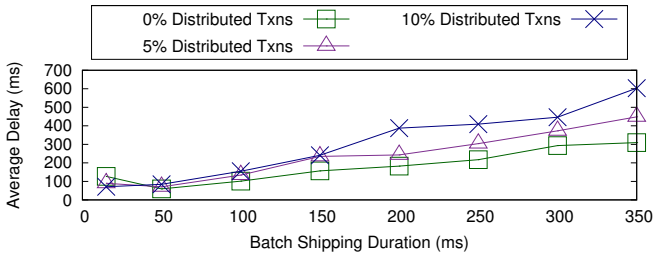


Figure 2.10: Study of Batch Shipping Duration

Study of Batch Shipping Frequency

We now analyze the interplay between batch shipping frequency and average delay in Janus. The evaluation is performed using the standard deployment with 120 clients. The results are shown in Figure 2.10. The transactional throughput numbers were similar to the ones observed in previous experiments, and hence, are not reported.

When no distributed transactions are present, as the batch shipping frequency decreases from 350 ms to 50 ms, the average delay reduces from 309 ms to 59 ms. The delay decreases because as the batch shipping period decreases, the batches are shipped more frequently and the overhead in ingesting smaller batches at the column partitions is not high enough to affect performance. But, when the batch shipping frequency is decreased further, the average delay for a transaction increases to 127 ms for a shipping frequency of 15 ms. This scenario results in MonetDB being hit with a high very update rate, which is not suited to its column-oriented design. Batch shipping frequency values below 50 ms result in higher delay values. A shipping frequency less than 50 ms does not

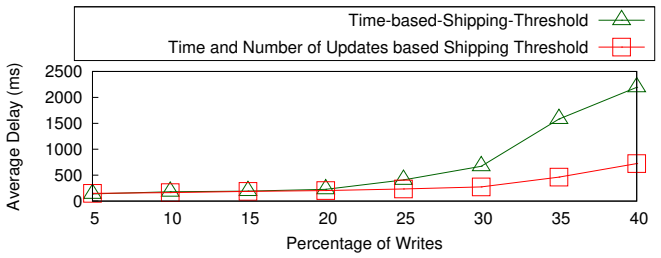


Figure 2.11: Combining Batch shipping Frequency with an update-based Shipping Threshold - 0% Distributed Txns

benefit from the effect of batch committing the changes at a column partition.

The presence of distributed transactions makes Janus more sensitive to a higher batch shipping duration. Distributed transactions lead to batch dependencies and a greater shipping period might result in longer chains of dependencies, resulting in higher average delay. In the case of 10% distributed transactions, increasing the shipping period from 300 to 350 results in the average delay value increasing by 35%. When no distributed transactions are present, the same increase in the shipping period only results in a 5% increase in average delay. On the flip side, dependencies resulting from distributed transactions also result in reducing the affect of decrease in shipping duration. Waiting for batch dependencies to arrive from other row partitions, reduces the high update rate resulting from a low shipping frequency value. In the case of 5% distributed transactions, decreasing the frequency from 50 ms to 15 ms, increases the delay at a slower rate as compared to the case with no distributed transactions. This shows that as the percentage of distributed transactions in the workload increases the batch shipping frequency value should be reduced.

These results illustrate that the optimal value of batch shipping frequency for our standard deployment of Janus is 50 ms. Janus can update the column partitions in as little as 59 ms with no distributed transactions and 85 ms with 10% distributed transactions.

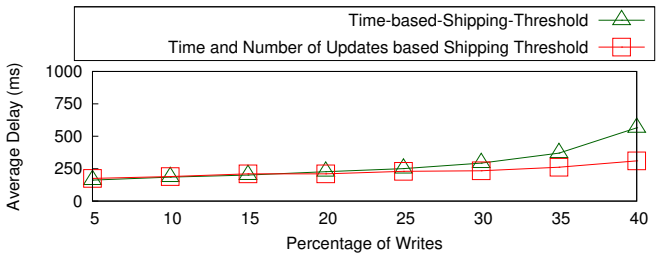
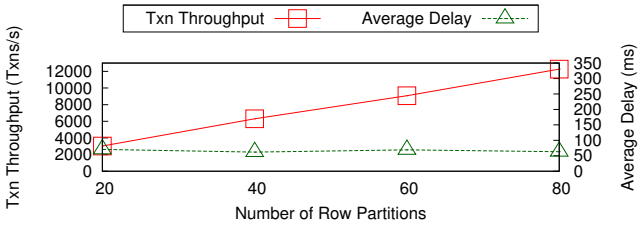


Figure 2.12: Combining Batch shipping Frequency with an update-based Shipping Threshold - 5% Distributed Txns

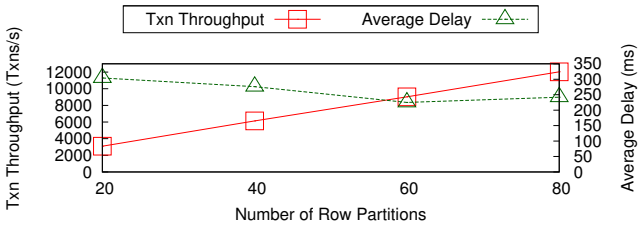
Addition of Update based Shipping Threshold

Section 2.4.4 discusses the bottleneck of using a time-based threshold for batch rollover, and how write-heavy workloads and long running transactions can lead to an increase in average delay. To mitigate the affect of only using a time-based threshold, Janus is integrated with the ability to specify an additional threshold to close a batch based on the number of updates (*batch shipping update-threshold*). We re-perform the experiment where the read-write ratio is varied (Figure 4.3), but with an additional batch shipping update-threshold, which is set to 100 updates. The threshold is set based on the number of updates present in a batch before the delay starts increasing (at 25% writes). Results in Figure 2.11 illustrate that adding an update-based threshold helps in reducing the increasing in delay as the percentage of writes increases.

Figure 2.12 shows the affect of update-based threshold in the presence of distributed transactions. We observe that even with distributed transactions, the update-based threshold helps in reducing the increase in size of batch sizes as percentage of writes increases. This leads to a slower increase in delay when compared to the case where only a time-based threshold is employed. The absolute delay increases slowly with distributed transactions (as the percentage of writes increase), because the dependencies among the distributed transactions result in ingestion of multiple batches together and reducing the write-activity on MonetDB, which leads to avoiding queuing of batches at the column



(a) Batching Shipping Frequency: 50 ms



(b) Batch Shipping Frequency: 250 ms

Figure 2.13: Scaling Out Janus

partitions.

Scaling Out

Figure 2.13 illustrates the performance of Janus while scaling-out. The percentage of distributed transactions was set at 5%. The number of clients invoked per row partition is kept constant throughout the experiment, and increase from 120 with 20 row partitions to 480 with 80 row partitions. This ensures that the amount of contention would be the same over the entire experiment. The ratio of row to column partitions is also kept constant at 2:1, and the execution engine servers are scaled from 4 to 16. Evaluation is performed with batch shipping frequency of 250 ms as well as 50 ms (time-based threshold only). As Janus scales-out the number of partitions, throughput increases linearly and the average delay remains constant in the range of 65-70 ms with 50 ms shipping frequency, and 250-300 ms with a shipping frequency of 250 ms. These numbers illustrate that Janus's architecture is capable of scaling-out, while still updating the column partitions in real-time.

Performance of Analytical Queries

For studying analytical performance, aggregate queries calculating the average and minimum value of an attribute were run on both Janus and the baseline setup. A single column partition query calculating the average with a filter predicate took 4.6 ms on Janus and 23.2 ms on the baseline setup. A query to compute the minimum took 1.80 ms on Janus and 13.2 ms on the baseline setup. Janus performs better for analytical queries as it executes such queries on a column-oriented design, which is more suited for these queries than the row-oriented design employed by the baseline setup. As multiversioning support is not provided by MonetDB, nor by any open source column database, we did not evaluate the execution of multi-column partition queries (Section 2.5.2).

2.7 Related Work

Janus is motivated by and related to a wide range of research in the areas of hybrid OLTP-OLAP systems, hybrid storage layouts, data shipping and database replication.

Hybrid OLTP-OLAP Systems. Most of the proposed solutions supporting both OLTP and OLAP workloads, execute on a single server [52, 42, 53]. These solutions do not scale out, which is an essential requirement for a cloud datastore. Furthermore, most of these systems [54, 52, 55] are tailored to a main-memory design.

Hyper [54] handles both OLTP and OLAP workloads in either a column or a row-oriented *main-memory* environment, using the operating system's copy-on write technique. Virtual memory snapshots are created when read-only OLAP queries arrive. Hyper's performance is dependent on efficient forking during copy-on-write. Plattner [56] proposes supporting both transactional and analytical processing using *in-memory* column stores. Based on this vision, SAP has introduced SAP HANA [55], a main-memory engine for supporting both transactions and analytics.

Some systems have proposed adaptive layout transformation to support both OLTP and OLAP workloads in a single server setting. Arulraj et al. [53] use row, columnar and hybrid row-column representations (where a vertically partitioned group of columns are stored together) and then dynamically transform the storage layout among these representations. A multi-versioning engine is employed and each tuple is inserted in row-oriented layout. A background mechanism monitors the queries and transforms the layout by determining the attributes that are accessed together, and then grouping them. H_2o [57] also chooses from among multiple representations and a foreground layout transformation between the row, column, hybrid row-column representations. Although layout transformation can also provide the ability to take advantage of multiple representations, its performance is dependent on accurate workload prediction and can be susceptible to sudden workload shifts.

ES-2 [58] supports partitioning and stores data in a distributed file system. It uses distributed indexing and parallel sequential scans to answer OLAP queries, in addition to supporting OLTP workloads using a multiversion timestamping approach. It relies heavily on indexing and can lead to poor performance for ad hoc queries which access non-indexed data residing on the distributed file system. Unlike ES-2, Janus executes transactions and analytics on different servers as well as different representations, that are more suited for the respective workloads. ES-2 is based on a shared-storage architecture, whereas Janus employs a shared-nothing architecture.

Snappy Data [59, 60] integrates a distributed *in-memory* data-store (GemFire) engine with Apache Spark's runtime and provides support for OLTP, OLAP and streaming. Data can be stored in *either* row or columnar form. OLTP operations are supported using the in-memory engine and OLAP operations are supported using Spark's executors. Such an architecture removes the overhead of maintaining multiple data representations. However, it cannot provide separation of transactions and analytics on different servers

or different representations. Furthermore, since such an architecture uses a single data representation, it has to choose one right partitioning strategy for all workloads.

Storage in Multiple representations. Fractured Mirrors [43] was one of the first systems to store data in multiple representations. A copy of the data is stored in both the DSM (Decomposition Storage Model) layout [25], where each column is stored separately, and the row layout. A differential file is used for updating the columns. Janus is inspired from this approach and employs storage of replicas in multiple representations in a distributed setting, with differently partitioned representations. OctopusDB [61] also provides the ability to store data in multiple representations. Updates are executed by appending to a log, which acts as a primary copy of the data and multiple storage views can be incarnated in different representations. Other systems have also explored storing data in hybrid representations [42, 62, 63, 41, 21, 64, 65]. However, these approaches do not provide support for data partitioning and distributed transactions.

Column-Oriented Storage. Column stores are used for supporting OLAP workloads. MonetDB [14], Druid [66] and Vertica [26] are some available column-oriented DBMS, optimized for high volumes of query processing. C-Store [16] uses a columnar representation and is optimized for read-mostly workloads. C-Store has a write store for supporting transactional updates and a tuple mover for moving the data into the read store. The write store is also columnar and stores data in memory. The write store is limited in size, is co-located with the read store and is designed to optimize for workloads with low update rates. Some approaches aim to efficiently update column stores to support transactions [67, 68]. In contrast to Janus, these systems are primarily optimized for analytics, but are not designed to support OLTP workloads, which have a high rate of transactional updates.

Change Data Capture [27], Pub-Sub systems [44, 69] log shipping techniques [41, 70] and some recent ETL tools [71] have been used for continuously consuming changes

from transactional systems at downstream analytical systems. This approach is amenable to scaling out and provides separation of transactions and analytics. However, such approaches provide consistency guarantees only per partition. A mechanism to overcome this is to use a single log as a source of truth. The head of the log can become a source of contention in approaches using a single distributed commit log for supporting transactions. A recent scale-out approach from SAP HANA [72] also uses a distributed commit log to support both OLAP and OLTP workloads.

Database Replication. Many replication techniques have been developed to improve the read performance of database systems, while providing some level of replica consistency guarantees. Techniques providing strong consistency [73], and 1-copy Snapshot Isolation[74] guarantees have been developed. All these systems use the same representation for the replicas. On the other hand, Janus uses different representations, that update at different rates, and also aims to attain high transactional throughput. Hence, Janus updates the read-oriented representation in batches. Due to this, the read-oriented replica can lag from the primary update-oriented copy, but provides a transactionally consistent snapshot of the data. Akal et al. [75] describe a replication technique which enables updating in batches and executing 1-copy serializable read-only transactions with given freshness guarantees. Their replication technique can support different partitioning strategies for update and read-only copies. However, it uses a global log for recording updates, which can be a bottleneck for throughput. In addition, distributed transactions are not supported.

Real-time Analytics. Recently, several architectures have been proposed for supporting both event-based processing and real-time analytical queries [76, 77, 78]. These systems do not handle transactional updates and hence, do not support OLTP workloads. Lazybase [79] uses a combination of batching and pipelined updates, and provides a trade-off between read query latency and data freshness. AIM [77] uses a PAX [80]

like representation, where records are grouped into buckets and each bucket is stored column-wise. It employs differential updates and shared scans to support event-based processing and analytics in a scale-out setting. Lambda Architecture[78] stores data in two layers, a batching layer (like HDFS) optimized for batch processing and a speed layer, like a stream-processing engine, which processes data streams in real time. Each ad-hoc query is sent to both representations, and the results are then integrated.

2.8 Summary

In this chapter, we propose Janus, a hybrid, partitioned, multi-representation datastore, for handling diverse workloads and enabling consistent real-time analytics. An instance of Janus is designed to support OLTP and OLAP workloads. Janus supports transactional requests on row-oriented storage and uses an in-memory redo log inspired batching technique to capture the transactional changes. Janus then employs a graph-based dependency management algorithm to ingest the transactional changes at the column-oriented storage, which supports analytical queries. The devised data movement pipeline for creating, shipping and ingesting batches ensures that updates get incorporated at the column partitions with minimal delay. The data movement pipeline supports distributed transactions, as well as diversely partitioned representations. Evaluation with the transactional YCSB benchmark illustrates that Janus enables real-time analytics, while scaling-out, and without affecting transactional throughput.

Chapter 3

Typhon: Consistency Semantics for Multi-Representation Data Processing

3.1 Overview

The growing variety in structure and access methodologies has resulted in the fall of "one-size-fits-all" paradigm. Application data, which used be managed by a single database, is now spread across multiple data-processing engines. Although this results in performance benefits, it can led to consistency violations for applications.

An example of diverse application data being spread across representations is a Social Network. Consider a social network, consisting of users, with attributes and each user having a list of friends. The user attributes can be stored in a key-value store and friend relationships are stored in a graph database, to take advantage of the different characteristics of user attribute data and the friendship graph. However, consistency semantics are usually provided only at the granularity of each datastore, and as described

in Section 1.2.1, the example scenario can lead to inconsistent executions. A deleted friend of a user might be able to view a new phone number, which was entered by the user after the friendship deletion. This is a consistency violation from the application user perspective, as the new phone number was added after the deletion of the friendship, and should not be accessible.

Apart from different structure and workload characteristics, another reason for storing parts of the application data in different representations, is the need for different access permissions and diverse origins of the data. Some applications need to access multiple data sources, managed by various departments within an organization, with different access protocols. Consider the following example of medical data spread across representations. One data representation stores metadata related to a patient, like name, age, address etc. Another representation stores the list of doctors who have access to patient data. This representation can only be accessed by the hospital and the patient. A third data representation stores prescriptions and lab results, which are only accessible to the patient and the doctors permitted in the second representation. Similar to the social network scenario, if the consistency guarantees are only provided at the granularity of individual representation, it can lead to semantically inconsistent executions. For example, we can have scenarios where a doctor deleted by a patient is able to add prescriptions for the patient.

Enforcing consistency semantics at the granularity of each representation might provide performance benefits for the application, as no coordination is needed across representations. However, this consistency model is difficult to comprehend for the end user of the application and can have undesirable consequences in certain situations, as described above. To avoid application anomalies like to ones described above, we need to provide consistency guarantees for data spread across multiple representations. However for building such solutions and understanding their consistency guarantees, formal

consistency models are first needed to reason about the different techniques providing consistency across multiple representations of data.

In this chapter, we propose *Typhon,* a *multi-representation data processing framework*, which defines a novel *consistency model* for data stored and accessed in multiple representations. The data model in Typhon allows the application developer to express dependencies across different application data by defining logical *entities* to link data across representations. Each entity semantically links different *data items*. In the social network described above, the user tuple in the key-value store and the user node in the graph represent two different data items related to the same user entity. Defining a user as an entity enables Typhon to express data consistency semantics at the granularity of a user. By relating the operations on items belonging to the same logical entity, Typhon can capture dependencies across representations. Typhon's consistency model defines *implicit causal dependencies*, which capture the logical order intended by the application user. We formally define these implicit dependencies in the consistency model, and compare and contrast them to the traditional explicit dependencies between operations accessing items at a single representation.

We then design and present a protocol, *Cerberus,* which operates in Typhon's framework and provides consistency semantics for operations accessing data across representations. Cerberus targets social networking and other similar applications, where operations access data items related to a single entity, across multiple representations. Cerberus employs a *single-entity transaction model*, which allows each transaction to access multiple items related to a single entity, across multiple representations. Cerberus uses a lightweight mechanism based on version vectors to ensure that all the application requests satisfy implicit causal dependencies for the entity accessed in a transaction.

Next, we provide a formal description of Typhon's consistency model. Then, we introduce the single-entity model, and discuss various existing solutions that could be

employed, their shortcomings, and the trade-offs involved for designing a solution. We give an intuition for the designed protocol, Cerberus, and then describe the protocol, formally prove its guarantees and extensively evaluate it to study its performance.

3.2 Typhon's Multi-Representation Consistency Model

Typhon is a framework for multi-representation data processing. In this chapter, we also develop a protocol, Cerberus, which introduces a single entity transaction model intended for applications which do not require a general transaction model. In the single-entity transaction model, each transaction only accesses a single entity. Hence, the single-entity execution model is restrictive as compared to the traditional transactional model but more general than a simple key-value store model. Hence, we develop our theoretical multiple-representation consistency model in a more general setting, and this enables the formal model to cover a super-set of all the executions possible under Cerberus.

Application data in Typhon is stored in *n representations*, R_1, $R_2 \ldots R_n$. The database consists of *m entities*. Each entity may have information stored in different representations, in the form of *data items*. x_j refers to a data item corresponding to entity x in representation R_j. An entity may not have a data item in a particular representation, which implies that x_j can be null.

The data in different representations is accessed by transactions comprising reads and writes over a subset of data items. Each transaction, T_i consists of a begin, b_i, set of reads, $r_i(x_j)$, and writes, $w_i(x_j)$, followed by a commit, c_i. The operations of a transaction can span data items present in different representations. We assume each representation either provides serializable transactions or provides atomic read and write guarantees over the individual items. A *history* is defined as a schedule of begin, read, write and commit operations on items across all representations. The relation $<$ between

operations implies their order in the history.

Typhon models the dependencies between transactions at the granularity of the entities. This allows us to semantically link data items present at different representations. The dependencies can be divided into dependencies within a representation and those across representations. At a particular representation, any two operations accessing a data item, where one of the operations is a write, are defined to have a *conflict* and have an order among them. Additionally, we also define dependencies between transactions accessing data items related to the same entity, across representations. These dependencies capture the implicit order of operations on the same entity.

Conflict Graph. A conflict graph [81] captures the dependencies of a transaction on other transactions in a given history. If a conflict graph for a particular history is acyclic, the given history is serializable [81]. We now define the conditions for constructing the *conflict graph* in Typhon's multi-representational consistency model. We will then illustrate that the inconsistent semantic execution in the social network scenario mentioned in Section 1.2.1 and 3.1, cannot be captured in the traditional conflict graph model. We will also show that by adding an additional class of edges, Typhon's multi-representational consistency model is able to capture the inconsistent execution.

Transactions are the nodes in the conflict graph. Transactions accessing the same data item at a representation can lead to $w-w$, $w-r$ and $r-w$ edges. The conditions for adding these edges are the same as the conditions for adding such edges in the traditional conflict graph. There exists an edge from T_i to T_j, $T_i \rightarrow T_j$, if transactions T_i and T_j access the same data item x_k, and T_i precedes and conflicts with one of T_j's operations. $w-w$, $w-r$ and $r-w$ edges reflect the writes-after, reads-from and reads-before relationship between two transactions relative to a data item.

Implicit Causal Edges: Transactions accessing the same entity across representations can have implicit causal edges between them. These edges are added to capture

implicit causal dependencies, which arise on different items related to a single entity, but spread across representations. The conditions for adding implicit causal edges are below.

Definition 1 *There exists an implicit causal edge from T_i to T_j, $T_i \rightarrow T_j$, if transactions T_i and T_j access data items x_k and x_l of the same entity x, and $o_i(x_k) \in T_i$ and $o_j(x_l) \in T_j$ and $c_i < b_j$, where $o_i(x_k)$ and $o_j(x_l)$ are read or write operations and one of $o_i(x_k)$ or $o_j(x_l)$ is a write. T_i and T_j are defined to have an implicit causal dependency between them.*

The implicit causal edges capture the dependencies between transactions at the granularity of an entity. Typhon defines dependencies between transactions that have a "happens-before" relationship and have conflicting operations on a particular entity. If a transaction T accesses an entity, it is defined to be implicitly causally dependent on any transaction which committed before transaction T began, and has a conflicting operation on the corresponding entity. *The implicit causal dependency captures all the transactions that potentially might have affected transaction T.*

A cycle in the conflict graph, with one of the edges being an implicit causal edge, indicates that "happens-before" relationships have not been preserved. Hence, the addition of the implicit causal edges to the conflict graph model can be used to detect inconsistent semantic executions.

Inconsistency Example Description. We formally analyze the example of an inconsistent semantic execution order described in Section 1.2.1. Suppose, x is Alice's entity and y is Bob's entity. Key-value store and graph representations are represented as R_1 and R_2 respectively. x_1 refers to the data item storing Alice's information in the key-value store and x_2 is the data item storing Alice's information in the graph representation. Operations for removing Bob from Alice's friend-list, modifying Alice's phone number and accessing Alice's phone number are depicted by 3 transactions, T_1, T_2

and T_3 respectively.

$T_1 = b_1 w_1(x_2) c_1$ //Alice removes Bob from her friend-list

$T_2 = b_2 w_2(x_1) c_2$ //Alice changes her phone number

$T_3 = b_3 r_3(x_2) r_3(x_1) c_3$ //Bob reading Alice's phone number

In transaction T_1, deletion of Alice's friendship with Bob leads to modification of the edge between Alice's and Bob's node in the graph representation. As friend relationships are bi-directional, the deletion of Bob from Alice's list would also lead to a write on the item y_2, Bob's friendship information, but we omit it for brevity. Reading Alice's phone number in T_3, comprises a check to see if Bob is still a friend of Alice and if he is, then reading her phone number. T_3 is executed at corresponding representations. T_1 and T_3 arrive concurrently and T_2 starts after T_1 commits. Both representations guarantee atomic execution. Consider the execution of the semantically inconsistent history, H_1.

H_1: $b_1 b_3 r_3(x_2) w_1(x_2) c_1 b_2 w_2(x_1) c_2 r_3(x_1) c_3$

This execution results in a scenario where T_3 reads the older version of Alice's friend-list, before she deleted Bob. T_3 also reads her new phone number. This execution will lead to Bob reading the new phone number of Alice, which should not be allowed as Alice and Bob were not friends when she changed her phone number.

Consider the conflict graph of the history H_1 in Typhon's multi-representational consistency model, shown in Figure 3.1. Suppose, T_0 is a transaction initializing x_1 and x_2. First, consider the edges arising only out of conflicts at individual representations. Such edges are represented by solid edges in the figure. The graph is acyclic, when considering only these edges, and the execution of this history is permissible by the conflict graph. This shows that only considering the traditional conflict graph edges is not sufficient to capture inconsistent execution order over the data spread across representations.

Now, we add the implicit causal write-write edge, $\overline{w - w}$, arising from the implicit

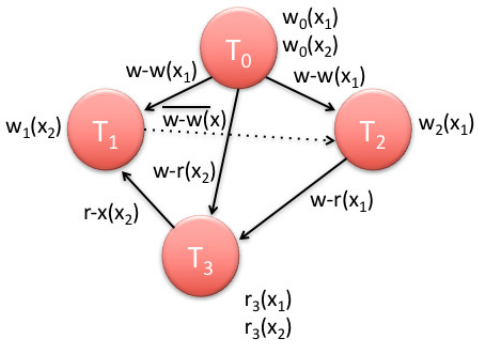


Figure 3.1: Conflict Graph showing inconsistent semantics

causal order between the writes on x_2 and x_1, performed by transactions T_1 and T_2 respectively. T_1 and T_2 access items related to entity x, $w_1(x_2) \in T_1$ and $w_2(x_1) \in T_2$, and $c_1 < b_2$ in H_1. Hence, from Definition 1, there is an implicit causal write-write edge, $\overline{w-w}$ from $T_1 \rightarrow T_2$. This edge is represented by the dashed line in the figure. The conflict graph has a cycle when this implicit causal edge is considered, because T_3's read operations do not obey the causal order between T_1 and T_2. This illustrates how the implicit causal edges help to detect the semantically inconsistent execution order.

For any particular execution, the presence of a cycle in Typhon's multi-representation conflict graph, with one of the edges being an implicit causal edge, indicates that the "happens-before" relationships have been violated. Analogous to a traditional conflict graph, Typhon's multi-representation conflict graph, can be used as a tool to reason about the consistency guarantees for applications storing and accessing data at multiple representations.

3.3 The Cerberus Protocol

Typhon's multi representation consistency model provides a theoretical framework to reason about the consistency of applications accessing data distributed across multiple

representations. We now design a protocol, Cerberus, which is targeted to social network and other such web applications, and preserves the implicit causal dependencies defined in Typhon. We first describe the application characteristics and the execution model suited for these applications, discuss the possible solutions and then describe Cerberus, the proposed protocol.

3.3.1 Application Characteristics

Social media and many other web-applications store their data in datastores that provide atomic guarantees only for a single key [82, 2, 83]. Some example of such datastores are graph stores like Tao [2] and Flock DB [84], key-value stores like Voldemart [82] and HBase [85], and document databases like Couchbase [83]. Such datastores are chosen since atomic operations are needed at the granularity of a single logical user entity. Such applications also have their data spread across many representations, due to different performance characteristics of representations, varied access permissions and diverse origins of datasets. These datastores provide only atomic read and atomic read-modify-write guarantees at a single data representation. However, these applications also need order-preserving guarantees over operations that access data related to a logical entity, but are spread over different representations. This implies that if there are conflicting non-concurrent operations on data items related to an entity in a particular order, all the operations should observe the order, even across representations. In a social network, if a user deletes a friend first and then changes their phone number, all application requests should observe that order. Another scenario where order of updates needs to be preserved, is a user reading the friend-list and writing a post on another users page, *if they are friends*. In the medical data scenario, an example is a doctor reading the prescription list of a patient, if he is in the permitted list of doctors for the patient. An

execution model and protocol only providing atomic guarantees at the granularity of a single item is not sufficient. The order of writes in the example of a user deleting a friend, and then changing the phone number, represents the “happens-before" relationship at the granularity of a user in the social network, which is the logical entity in this case. Applications accessing data across representations want such happens-before relationships to be preserved.

Additionally, since the representations store diverse data, the application semantics might not allow a user to write atomically across different representations. However, users expect the reads and writes to reflect the order of writes across different representations. In the social networking scenario described earlier, a user might not be able to atomically remove a friend and post a status update. But, if a user posts a status update after the deletion of the friend, they expect this order to be preserved by the application.

3.3.2 Single Entity Transaction Model

One possible solution to provide these guarantees is to use a traditional transactional model over all the data items. However, such a model is expensive and will incur an overhead of providing stronger guarantees than desired. As most of the application scenarios need transactional boundaries only at the granularity of a single entity, a general transactional model is not necessary. Additionally, since the application semantics in many cases do not provide the ability to atomically update data across representations in one user operation, a general transaction model might not be that useful. We need an execution model which allows the applications the ability to express the ordering of requests at the granularity of an entity.

To target the application scenarios described, Cerberus employs a *single-entity transaction model*. Clients access and modify data as transactions. Transactional boundaries

are only needed at the granularity of a single entity. Hence, transactions in Cerberus only access a single entity. Writes are restricted to a single representation, as such applications typically require atomic writes to a single representation based on the information in other representations, like writing a post on a users page based on the friendship status. The read operations access data items related to the same entity at different representations and the write operation updates the entity at a particular representation. A transaction executes in two phases. In the first phase, an application client performs all the reads. Writes, validation and commit are performed in the second phase. A read-only transaction can complete in a single phase. The single-entity transactional model allows the application to express user operations relating multiple data items present in different representations, without using a general-purpose transactional model. Protocols can be designed in such an execution model to provide guarantees only at the fine granularity of an entity.

3.3.3 Possible Solutions

Typhon's consistency model captures the "happens-before" relationships, which social-media and other such web-applications, want to preserve across data representations. These happens-before relationships are captured using implicit causal dependencies in Typhon. To cater to the above-mentioned applications, a protocol can be designed to support transactions in a single-entity execution model, and preserve the implicit causal dependencies across data representations. There are various techniques which can be employed to develop such a protocol and achieve the consistency guarantees defined in Typhon's model.

A simple technique would be to enforce a global order of operations across multiple representations. Deterministic schemes like Calvin [86] use a sequencer to pro-

vide a global order in partitioned databases. Such schemes can be adapted to a multi-representation setting. This is a restrictive solution as the global ordering enforces an order on all operations across all representations. This limits the underlying concurrency at each representation. Furthermore, scalable techniques to provide deterministic ordering [86, 87], which aim to attain more concurrency, support only a restrictive execution model, in which the read and write-sets of the transactions have to be known in advance. Another alternative to enforce a global ordering is to provide external consistency [31] on operations across multiple representations. However, solutions for external consistency either require access to expensive atomic clocks [31] or would need some other form of time synchronization across various representations [88]. Additionally, if multiple applications access data across these representations, a global ordering scheme can lead to one application's workload adversely affecting the performance of other applications. For example, a hot spot on certain data items in an application workload will affect the performance of all other applications.

To overcome the disadvantages of the global ordering schemes described, techniques can be built at the granularity of individual data items or at the granularity of Typhon's logical entities. One solution can be to use an order-preserving scheme like distributed strong strict two-phase locking (SS2PL) [81] over all the data items present in all representations. This would provide an order between non-concurrent transactions accessing an entity, and satisfy the guarantees in Typhon's consistency model. However, this would require lock managers, which can be scalability bottlenecks. Furthermore, a lock would have to be acquired for each operation. To reduce the amount of locks needed to be acquired per transaction, we can design the locking scheme at the granularity of the entity. Locking per entity would lead to ordering operations on a particular entity, and would order all conflicting operations on an entity. Although, this provides stronger guarantees than implicit causal order, it limits concurrency. Additionally, using pessimistic schemes

like locking would perform poorly in low-contention environments. To avoid these bottlenecks, we develop an optimistic scheme, that provides implicit causal order on operations accessing a single entity across all representations.

3.3.4 Cerberus Overview

Cerberus ensures that if there are any two transactions which are non-concurrent and have conflicting operations on the same entity, the order among these two transactions would be preserved even across representations.

The system architecture for employing Cerberus in Typhon's multi representational framework is illustrated in Figure 3.2. Cerberus uses a middleware approach, which has been widely used in partitioned and replicated environments [32, 89] In this approach, client requests are sent to a middleware layer, which is the Operation processing layer (OPL) in Cerberus's case. The OPL then routes requests to the corresponding data storage location(s) and collects the results and sends them back to the client. OPL also maintains metadata related to the data items and representations.

Each representation consists of a datastore layer which provides the isolation guarantees at the representation and the Cerberus layer, which preserves implicit causal order across representations. The protocol executes as a thin layer over the isolation guarantees provided at each representation. Each representation provides a key-value datastore interface, which supports single-key atomic read and test-and-set operations. Executing as a thin layer over the isolation scheme at each representation provides Cerberus the flexibility to augment existing datastores. We envisage the entities to be logical connections among the data at different representations, like a user for a social networking application. We believe that the entity identification should be done by the application developer, based on the application semantics.

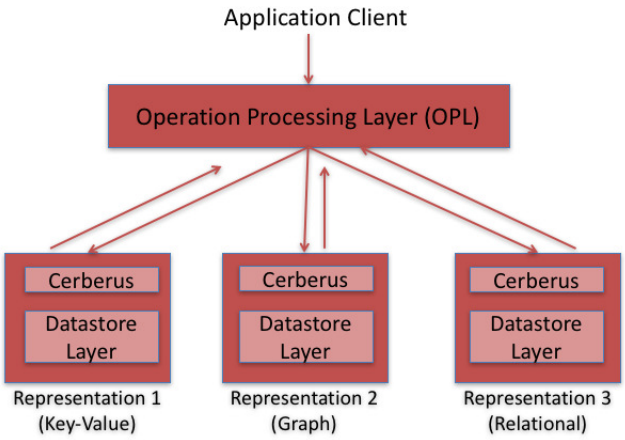


Figure 3.2: System architecture for Cerberus

For capturing Typhon's implicit causal dependencies, Cerberus stores version vector metadata at the granularity of the entity. The version vector metadata is stored both at the OPL and locally at each representation. The version vectors capture the extent of causal knowledge about the entities. Each transaction performs checks to ensure that the data item it is accessing has not been accessed by a conflicting transaction with higher causal knowledge about the entity.

Entity	Data Items	Global Version Vector (GVV)
x	$\{x_1\ x_2, x_3\}$	[0 1 3]
y	$\{y_1\ y_2, y_3\}$	[1 4 2]

Figure 3.3: Global Metadata in Cerberus

Data Item	Value	Update Version Vector (UVV)	Read Version Vector (RVV)
x_1	1	[0 1 2]	[0 1 3]
y_1	2	[1 4 2]	[1 4 2]

Figure 3.4: Metadata at each representation in Cerberus

3.3.5 Metadata Description

Each data item has a *version number* associated with it, represented as v_{x_i}, for data item x_i. The OPL maintains information about the data items associated with an entity and the latest version of these data items. The OPL maintains a *version vector*, that comprises the latest version number of all the data items associated to the entity. This global version metadata is referred to as the *global version vector* (GVV) of an entity. The GVV of an entity x comprises v_{x_1},v_{x_2}...v_{x_n}, where x_1, x_2 ... x_n are the data items of entity x and n is the number of representations. R_1 ... R_n are the corresponding representations. Figure 3.3 illustrates the GVV for two entities; each entity has data items associated with it at three representations.

Representations also store some metadata. Each representation maintains two local version vectors, an *update version vector* (UVV) and a *read version vector* (RVV) associated with each data item. The *UVV* of x_i stores a version vector corresponding to entity x. The i^{th} entry of x_i's UVV, represents the latest version of data item, x_i. All other entries of x_i's UVV represent the latest version of the data items, $x_1..x_n$, known at the representation R_i, at the time of x_i's last update. The *RVV* of x_i also stores a version vector corresponding to entity x. All the entries of x_i's RVV represent the latest version of the data items, $x_1..x_n$, known at R_i, at the time of x_i's last read. Figure 3.4 illustrates the local version vectors at a representation, R_1, for data items x_1 and y_1. All the entries in the version vectors are initialized to 0 at start-up and monotonically increase as updates and reads are processed.

3.3.6 Transaction Execution

Cerberus provides implicit causal ordering by performing causal validations, which compare the global and local version vectors. The global and local version vectors cap-

Algorithm 1: Transaction Execution at OPL

```
1  gvv - global version vector
2  Transaction T
3  function executeTransaction(t)
       // Transaction T accesses entity x
4      gvv(x, T) ← Read gvv of the entity x
5      for each representation i which has a data item in the read_set do
6          (read_result_i, version_read_i, read_status_i) =
7              read_rep_i(T, x_i, gvv(x, T))
8          if read_status = abort then
9              Abort the transaction
10             return
11         end
12     end
13     Determine write_set and new value_{x_i}
14     commit_status ← commit
15     if write_set is not null then
           // Item to write is at Representation i
16         (gvv_updated_version(x_i), commit_status) = write_rep_i(T,
17             x_i, value_{x_i}, gvv(x, T), version_read_i)
18         if commit_status = commit then
19             Update the global version vector with entry in
                gvv_updated_version(x_i)
20         end
21     end
22     return commit_status
```

ture the global and local version information known regarding the different data items related to an entity. Intuitively, for any operation accessing a particular data item in a transaction T, if the version entry corresponding to any representation in the local version vector of a data item is greater than the corresponding entry in the global version vector of the entity associated to the item, it means that a transaction causally ordered after T has modified or read the data item. In that case, the transaction is aborted. Otherwise, the causal validation passes successfully and transaction commits. We now give a detailed description of each phase of a transaction in Cerberus. The algorithms for

Algorithm 2: Read at Representation i

```
23 gvv - global version vector
24 rvv - local read version vector
25 uvv - local update version vector
26 function read_rep_i(T, x_i, gvv(x,T))
27     (x_i, val(x_i), uvv(x_i,T), rvv(x_i,T)) ← read_DS(T, x_i)
       // Causality check
28     if read_causal_check(gvv(x,T), uvv(x_i,T)) ≠ true then
29         return (null, null, abort)
30     end
       // Update read results and versions read
31     read_status = commit
32     read_result = val(x_i)
33     version_read = (uvv(x_i,T), rvv(x_i,T))
       // Check if read version vector needs updating
34     if is_rvv_upto_date(gvv(x,T), rvv(x_i,T)) ≠ true then
35         write_set_tuple = (x_i, val(x_i), uvv(x_i,T), gvv(x,T))
36         version_read = (uvv(x,T), gvv(x,T))
           // Update rvv via Test-and-set
37         read_status = commit_DS(T,
38             x_i, version_read, write_set_tuple)
39     end
40     if read_status = commit then
41         return (read_result, version_read, commit)
42     end
43     else
44         return (null, null, abort)
45     end
46 function read_causal_check(gvv(x,T), uvv(x_i,T))
47     return gvv(x,T) ≥ uvv(x_i,T)
48 function is_rvv_upto_date(gvv(x,T), rvv(x_i,T))
49     return rvv(x,T) ≥ gvv(x_i,T)
```

Algorithm 3: Write at Representation i

```
50 function write_rep_i(T, x_i, value_{x_i},
51       gvv(x,T), version_read)
52     if version_read is null then
53         (x_i, val(x_i), uvv(x_i,T), rvv(x_i,T)) ← read_DS(T, x_i)
54         version_read = (rvv(x_i,T), uvv(x_i,T))
55     end
       // Causality check
56     if write_causal_check(gvv(x,T), uvv(x_i,T), rvv(x_i,T)) ≠ true then
57         return (null, abort)
58     end
       // Compute new local update version vector
59     (new_uvv(x_i), gvv_updated_version(x)) < −
60         get_new_uvv(i, gvv(x,T))
61     write_set_tuple = (x_i, value_{x_i},
62         (new_uvv(x_i,T), rvv(x_i,T))
       // Commit via Test-and-set
63     commit_status = commit_DS(T, x_i, version_read, write_set_tuple)
64     return (gvv_updated_version(x), commit_status)
65 function write_causal_check(gvv(x,T),
66       uvv(x_i,T), rvv(x_i,T))
67     return gvv(x,T) ≥ uvv(x_i,T) & gvv(x,T) ≥ rvv(x_i,T)
68 function get_new_uvv(i, gvv(x,T))
69     for each version in the version vector gvv(x,T) do
70         if version! = i then
71             new_uvv(x_i)[version] ← gvv(x,T)[version]
72         end
73         else
74             new_uvv(x_i)[version] ← gvv(x,T)[version] + 1
75             gvv_updated_version(x)[version] = gvv(x,T)[version] + 1
76         end
77     end
78     return (new_uvv(x_i), gvv_updated_version(x))
```

Algorithm 4: Key-Value Datastore Interface

```
79 function read_DS(t, x_i)
80 function commit_DS(t, x_i, version_read, tuple_to_write)
```

read and write phase are presented in Algorithms 1, 2 and 3 respectively. The interface exposed by the datastore layer of each representation is shown in Algorithm 4.

Read Phase: The application client sends the read-set of the transaction to the OPL, comprising the data items to be read. When a transaction, T, arrives at the OPL, it reads the GVV of entity x corresponding to the transaction, denoted as $gvv(x, T)$. The GVV of an entity is only read once during a transaction. The OPL then sends the read requests to the corresponding representations, where the data items are stored. The request to a representation comprises the data item to be read and the GVV of the entity related to the corresponding data item. The read request is sent to the Cerberus layer at each representation, which then sends the request to the datastore layer of the representation. The datastore layer atomically reads the value as well as the metadata associated with the data item. The metadata comprises the UVV and the RVV associated with the data item, referred to as $uvv(x_i, T)$ and $rvv(x_i, T)$, the UVV and RVV of x_i for transaction T. The Cerberus layer then performs a check, defined as a *causal read check* to validate that the read data item, x_i, satisfies the causal dependencies related to entity x, across representations (Algorithm 2-Line 46). The causal read check compares the GVV of the entity to the UVV of the data item to validate that the read operation follows the implicit causal order. For a read of data item x_i to be valid, the GVV of the corresponding entity, x, $gvv(x, T)$, should be greater than or equal to the UVV associated to the data item, $uvv(x_i, T)$. If the read does not pass the causal read check, then the transaction is aborted and the decision is sent to the OPL. Otherwise, the read operation is successful.

If the read is successful and the RVV of the data item is less than the GVV of the transaction, then the RVV of the data item, $rvv(x_i, T)$ is updated to the GVV, $gvv(x, T)$ (Algorithm 2-Line 35). The RVV is updated by sending a request to the datastore layer. The datastore layer uses an atomic test-and-set operation to update the data item. The

test-and-set ensures that the RVV of a data item is updated only if the item is not concurrently updated after the read validation. This guarantees that the causal read check is still valid. If the read is successful, then the read result comprises the value of the data item read and the RVV and UVV of the data item. The OPL collects the read results from all representations and sends them to the application client. Note that RVV is only updated by a read when there has been an update to any representation associated with the entity, after the most recent read of the data item. Hence the protocol has minimal overhead for a read-heavy workload.

Write Phase: If the read phase is successful and the transaction is read-only, then the transaction is committed. Otherwise, in the second phase, the application client sends the write request to the OPL. The OPL then sends the write to the representation to be modified. As noted above, writes are performed on a single representation. Each write request contains the data item, x_i, the value to be written, $value_{x_i}$, and the GVV of the entity x read by the transaction, $gvv(x, T)$. If the transaction performed a read phase at the representation to which the write-request is being sent, then the request also contains the local version vectors, the UVV and the RVV of the data item read during the read-phase. If the data item, x_i was not present in the read set, then the Cerberus layer first sends a read request to the datastore layer. This action is performed to access the local version vectors, RVV and UVV corresponding to the data item x_i at the particular representation. The RVV and the UVV are needed to validate if the write operation follows implicit causal ordering. Similar to the read phase, the GVV, $gvv(x, T)$ is compared against the local version vectors (UVV and RVV) of x_i, to verify that the write operation follows the causal order across representations (Algorithm 3-Line 65). This check is defined as a *causal write check*. Since implicit causality in Typhon implies that a write operation on an entity depends on preceding reads and writes, causal write check verifies whether the GVV is greater than or equal to both the respective UVV,

as well as the RVV. If the RVV is greater, it means that a transaction causally ordered after the current transaction, has *read* the data item. On the other hand, if the UVV is greater, it means that a transaction causally ordered after the current transaction, has *written* the item. If the causal write check is successful, then the commit phase is processed for the transaction. If the check fails, the transaction is aborted.

Commit Phase: If the write obeys causal relationship across representations, the commit phase is performed (Algorithm 3-Line 63). During this phase, the datastore layer updates data item. The datastore layer uses an atomic test-and-set operation to update the value and the corresponding metadata related to the data item. The test-and-set verifies that the RVV and the UVV have not been modified after the causal read and write checks. If the RVV was updated during the read-phase by the current transaction, the test-and-set operation compares with the updated RVV. This mechanism ensures that the causality checks performed are still valid. Along with writing the new value of the data item, the UVV of the data item in the write-set is also updated. The new UVV for the modified data item, x_i, is equal to the GVV of entity x, corresponding to the transaction, $gvv(x, T)$, except the entry for x_i (Algorithm 2-Line 68). The i^{th} entry in $uvv(x_i)$, corresponding to x_i at R_i, is generated by monotonically incrementing the previous version number of x_i at R_i.

Post Commit Processing: The commit decision is then sent to the OPL. If the commit is successful, the new version number of the modified data item is sent back to the OPL, along with the commit decision. If x_i is the updated data item, the latest version of data item x_i is sent. The OPL then updates the GVV for the entity modified by the transaction to reflect the updated data item. After the GVV is modified, the commit decision is sent back to the application client. If the transaction is aborted, the OPL just returns the decision to the client, without performing any updates to the GVV.

Cerberus would not allow the inconsistent execution described in Section 3.2. Causal

read check for transaction T_3 would fail at the key-value representation, as the local version entry in the UVV of x_1 corresponding to the graph representation would be greater than the corresponding entry in T_3's GVV. T_3 would then be aborted. The local version entry in the UVV of x_1 would be greater as the change of phone number in T_2 would update the UVV of x_1 with the updated entry corresponding to version information of x_2, as T_2 starts after T_1 commits. Information flows through updates of the global and local version vectors, and the causal reads and write checks in Cerberus ensure correctness. Cerberus provides a lightweight version-vector based approach, to provide implicit causal ordering for transactions accessing data across representations.

3.4 Cerberus: Proof of Correctness

Section 3.2 discusses the construction of the conflict graph in Typhon's multi-representation framework. Our protocol, Cerberus, ensures that the conflict graph does not have any cycle, which involves an implicit causal edge. Since the implicit causal edges capture the causal dependency across representations, this guarantee ensures that all transactions in Cerberus obey implicit causal dependencies. The comparison of global version vector and local version vectors for each operation guarantees that causal order is obeyed across representations. Cerberus employs a causal read check (Algorithm 2-Line 46) for each read operation and a causal write check (Algorithm 3-Line 65) for each write operation. The checks in combination with the atomic guarantees from the datastore layer ensure that there are no cycles involving an implicit causal edge.

Cerberus orders *any* operation on a data item using the global version vector (GVV) of the entity accessed by corresponding transactions. The following lemma captures this property.

Lemma 1 *If the conflict graph in Typhon's consistency model has an edge from $T_j \rightarrow T_p$, then Cerberus ensures that the global version vector (GVV) accessed by T_p will be greater or equal to the global version vector (GVV) accessed by transaction T_j, $gvv(x, T_p) \geq gvv(x, T_j)$, where x is the entity accessed by transactions T_j and T_p.*

Proof: Consider all possible edges in the conflict graph in Typhon's multi-representation consistency model. We will show that for each case the GVV of T_p will be be greater than or equal to the GVV accessed by T_j. Note that since each transaction in Cerberus accesses a single entity, if there is an edge between two transactions, it implies that both such transactions must access the same entity.

- *$T_j \rightarrow T_p$ is a $w - w$ edge.* Suppose the $w - w$ edge exists due to operations on item x_k. The causal write check in Cerberus ensures that the GVV corresponding to T_p will be greater than or equal to the update version vector (UVV) of the data item x_k accessed by T_p. Additionally, as T_j updates the data item x_k, it will also update the UVV (Algorithm 2-Line 68), which is then read by T_p. The entry corresponding to version number of item x_k is updated. The update of the UVV is executed during the commit phase of T_j and takes place atomically with the update of value of x_k. The UVV is updated to be greater than the GVV accessed by T_j. This ensures that the UVV accessed by T_p is greater than the GVV accessed by T_j.

  $gvv(x, T_p) \geq uvv(x_k, T_p) > gvv(x, T_j)$, where x_k is a data item belonging to entity x and accessed by T_j and T_p.

- *$T_j \rightarrow T_p$ is a $r - w$ edge.* Suppose the $r - w$ edge exists due to operations on item x_k. The causal write check in the protocol ensures that the GVV corresponding to T_p will be greater than or equal to the read version vector (RVV) of the item x_k. The atomic test-and-set of the RVV (Algorithm 2-Line 35), during the read

validation of T_j, ensures that the RVV accessed by T_p will be greater than or equal to the GVV of T_j.

$gvv(x, T_p) \geq rvv(x_k, T_p) \geq gvv(x, T_j)$, where x_k is a data item belonging to entity x and accessed by T_j and T_p.

- $T_j \rightarrow T_p$ *is a* $w - r$ *edge.* Suppose the $w - r$ edge exists due to operations on item x_k. As T_p reads the item x_k, the causal read check during the read phase of T_p ensures that the GVV accessed by T_p will be greater than or equal to the UVV of the item x_k. Whereas, the atomic update of the UVV (Algorithm 2-Line 68) during the commit phase of T_j ensures that the UVV after T_j's write (read by T_p) is greater then the GVV corresponding to T_j.

  $gvv(x, T_p) \geq uvv(x_k, T_p) > gvv(x, T_j)$, where x_k is a data item belonging to entity x and accessed by T_j and T_p.

- $T_j \rightarrow T_p$ *is an implicit causal edge.* Suppose the implicit causal edge exists due to operations on an entity x. By Definition 1, the commit of T_j precedes the begin of transaction T_p, $c_j < b_p$. This condition ensures that T_j will access an older version of the GVV, as compared to T_p. As version vectors monotonically increase over time, the GVV read by T_p will be greater than or equal to the GVV corresponding to T_j.

  $gvv(x, T_p) \geq gvv(x, T_j)$

Considering all the different types of edges between T_j and T_p, we have the condition:

$$gvv(x, T_p) \geq gvv(x, T_j) \tag{3.1}$$

Hence, if there is an edge in the conflict graph from $T_j \rightarrow T_p$, then the GVV accessed by T_p will be greater than or equal to the GVV accessed by T_j.

■

Lemma 1 establishes that any edge in the conflict graph corresponds to a non-decreasing change in the GVV of the entity corresponding to the edge. However, we still have to show that there cannot be a cycle with all the transactions in the cycle having the same GVV. We will now use Lemma 1 and the property of each edge of the conflict graph in Cerberus from the proof of Lemma 1, to prove that the conflict graph can not have a cycle involving an implicit causal edge.

Property 5 *If G is a conflict graph generated by an execution using Cerberus, then G would not have a cycle involving an implicit causal edge.*

Proof:

To prove that the conflict graph does not have any cycles involving an implicit causal edge, lets first suppose that there is a cycle in the conflict graph, and one of the edges in the cycle is an implicit causal edge. Consider the conflict graph and suppose $T_1 \rightarrow ..T_i \rightarrow T_j \rightarrow ..T_n \rightarrow T_1$ is the cycle and $T_i \rightarrow T_j$ is an implicit causal edge. As $T_i \rightarrow T_j$ is an implicit causal edge, T_i and T_j have a $\overline{w-w}$, $\overline{w-r}$ or a $\overline{r-w}$ edge between them (Section 3.2).

First, lets consider that $T_i \rightarrow T_j$ is either an **implicit causal writes-after $\overline{w-w}$ or an implicit causal reads-from edge, $\overline{w-r}$**. Suppose T_i writes data item x_l. Given that $T_i \rightarrow T_j$ is an implicit causal edge, $c_i < b_j$ (Definition 1). As T_i writes x_l, it will update the entry corresponding to the version number of x_l in the GVV of entity x. As T_j begins after T_i commits, and T_i updates the GVV during the commit phase (Algorithm 1-Line 19), hence, the GVV read by T_j will have a higher version number in the entry corresponding to x_l as compared to that in the GVV corresponding to transaction T_i.

Note that a transaction reads the GVV during the beginning of the transaction.

$$gvv(x, T_j) > gvv(x, T_i) \tag{3.2}$$

Now, applying the Lemma 1 transitively to nodes across each edge of the cycle, $T_j \rightarrow T_p..T_n \rightarrow T_1..T_i \rightarrow T_j$:

$$\begin{aligned} gvv(x, T_i) \geq gvv(x, T_1) \geq gvv(x, T_n) \geq gvv(x, T_p) \geq gvv(x, T_j) \\ \implies gvv(x, T_i) \geq gvv(x, T_j) \end{aligned} \tag{3.3}$$

Since, both the Equations 3.2 and 3.3 cannot be true simultaneously, we have arrived at a contradiction. A cycle in the conflict graph can not involve a $\overline{w-r}$ or a $\overline{w-r}$ implicit causal edge.

Next, consider the case that $T_i \rightarrow T_j$ is an **implicit causal reads-before edge,** $\overline{r-w}$. T_i does not write an entity x, otherwise $T_i \rightarrow T_j$ would be an implicit writes-after, $\overline{w-w}$, or reads-from edge, $\overline{w-r}$. Since, T_i only reads entity x, it must have an incoming reads-from edge in the cycle. Consider such an edge, $T_h \rightarrow T_i$. This edge can either be a traditional reads-from edge, $w-r$ or an implicit causal reads-from edge, $\overline{w-r}$.

- *$T_h \rightarrow T_i$ is a $w-r$ edge on item x_k.* From the discussion about the $w-r$ edge in proof of Lemma 1, we have:

  $gvv(x, T_i) \geq uvv(x_k, T_i) > gvv(x, T_h)$, where x_k is a data item belonging to entity x.

- *$T_h \rightarrow T_i$ is an implicit causal reads-from edge, $\overline{w-r}$ on entity x.* As $T_h \rightarrow T_i$ is an implicit causal reads-from edge, using Equation 3.2, the GVV accessed by T_i would

be greater than the GVV accessed by T_h.

$$gvv(x, T_i) > gvv(x, T_h)$$

Considering $T_h \rightarrow T_i$ as both a $w - r$ or $\overline{w - r}$ edge, we have the condition that the GVV accessed by transaction T_i would be greater than the GVV accessed by transaction T_h:

$$gvv(x, T_i) > gvv(x, T_h) \tag{3.4}$$

Applying Lemma 1 transitively across each node in the cycle, $T_i \rightarrow T_j$ .. T_n .. T_1 .. $T_h \rightarrow T_i$, as done in Equation 3.3, we have:

$$\begin{aligned} gvv(x, T_h) \geq gvv(x, T_1) \geq gvv(x, T_n) \geq gvv(x, T_j) \geq gvv(x, T_i) \\ \implies gvv(x, T_h) \geq gvv(x, T_i) \end{aligned} \tag{3.5}$$

Considering Equations 3.4 and 3.5, we arrive at a contradiction. Hence, a cycle with an implicit $\overline{r - w}$ causal edge cannot exist.

Combining both cases, there can not be a cycle in the conflict graph which involves a $\overline{w - w}$, $\overline{w - r}$ or $\overline{r - w}$ implicit causal edge. Since, these are the only types of implicit causal edges, this illustrates that the conflict graph can not be involved in a cycle with any of the edges being an implicit causal edge. This ensures that Cerberus provides implicit causal order on logical entities across representations.

■

3.5 Scaling and Recovery in Cerberus

3.5.1 Scaling

Cerberus handles the large scale of the data by *scaling-out* via data partitioning. Both the data and the metadata at the OPL can be partitioned. Data at each representation can be partitioned among multiple servers. The OPL has information about the location of different data items at a particular representation and routes each read or write request accordingly to the appropriate representation server. Additionally, to handle a high volume of requests, Cerberus can scale-out by using multiple OPL servers. The logical entities are then partitioned among the multiple OPL servers. As each transaction only accesses a single entity, each transaction is processed at a single OPL server. The OPL and the representation servers can be scaled independently of each other. Cerberus's architecture and execution model enables it to seamlessly scale to handle the large-scale of the data.

3.5.2 Recovery

Cerberus can recover from crash failures of both representations as well as OPL servers. Each representation maintains and persists its own redo log. The datastore layer of a representation ensures that each committed operation is written to the redo log. Note that the updates to the RVV during reads are treated the same way as updates to the items, and are also logged. This redo log can then be replayed to recover each representation after a crash failure. A crash failure can occur after a write transaction has committed at a representation and before the latest version of the data item was sent to the OPL. Hence, on recovery, each representation sends the latest version of its data items to the OPL. On the failure of a representation, reads and writes to other

representations can still proceed. Each transaction reads the latest GVV of the entity from the OPL and if the transaction performs a write, it updates the GVV before sending the commit to the application client. Hence, as long as the OPL is available, transactions only accessing data from available representations can proceed.

If the OPL fails, Cerberus becomes unavailable. If the metadata is partitioned among multiple OPL servers, then on the failure of an OPL server, only the transactions on the entities maintained by the failed server cannot proceed.

To recover after crash failures, the OPL server(s) has to contact the appropriate data representations. Each representation then sends the latest version of all the data items to the OPL. Each OPL server uses this information to update the GVV of all the entities it maintains. OPL only maintains the version vector metadata in main-memory. This helps in reducing the commit latency but increases the time of recovery of the OPL. The failure of an OPL server only affects liveness, and does not affect correctness. The OPL can be made highly available by replication using Paxos [29] or other replication mechanisms.

3.6 Evaluating Cerberus

We compare and contrast the performance of Cerberus against both weak and strong consistency providing schemes, and study its performance under varied number of clients, different access distributions and read-write ratios and when scaling OPL servers. We first give a brief description of the benchmark and baselines used, describe our evaluation setup, and then discuss the experimental results.

3.6.1 Benchmark Description and Baselines Used

We develop a transactional micro-benchmark based on the YCSB benchmark [48] to issue operations accessing data across multiple representations. The YCSB benchmark is used to benchmark Cloud databases, and issues single-key read and write operations. Our micro-benchmark issues transactions which follow the single-entity transaction model presented in Section 3.3.2. Each transaction groups multiple single-key read and write operations on data items related to a single entity. The micro-benchmark issues different types of transactions discussed below.

Read-only transactions read 2 data items related to an entity, at 2 different representations.

Read-write transactions read a data item and then write another data item related to the same entity. 50% of the read-write transactions have the read and write on the same representations. In the other 50% of the cases, the read and the write are present on different representations.

Write-only transactions: Each such transaction writes a data item related to an entity present on a particular representation.

3.6.2 Baselines.

We compare Cerberus with three contrasting baselines, one providing weak consistency guarantees and the other two providing strong consistency guarantees on operations accessing data across representations.

Single-Key Atomicity scheme executes atomic read or write operations, and provides no consistency guarantees on operations accessing data across representations.

Distributed Strong Strict Two-Phase Locking (Distributed SS2PL) [81] is a commit order-preserving concurrency control protocol which ensures that the commit or-

der between all the non-concurrent transactions is preserved. Hence, distributed SS2PL satisfies the consistency guarantees across representations defined in Typhon. Each transaction first acquires locks on all the items its accesses. All the locks (both read and write locks) are released at the end of the transaction.

Locking Per Entity scheme provides a serializable order of transactions. The Locking per entity scheme also satisfies the consistency guarantees defined in Typhon. An operation accessing a data item related to an entity leads to a lock on the entire entity. This orders all the transactions that access a particular entity. Locking the entity provides the advantage that if a transaction reads data from multiple representations, it only has to acquire a single lock.

We do not compare with the multi-representation adaption of deterministic ordering schemes like Calvin [86] or Bohm [87], because they need the read and write sets of a transaction to be known apriori. Cerberus uses an execution model where the transactions are executed in an ad-hoc manner. First, the client sends read requests and then based on the results returned, can decide whether or not to send a write request, and which data item related to the entity to write.

3.6.3 Experimental Setup

The experiments were performed on a local cluster. Each machine in the cluster runs a 8-core Intel Xeon E5620 processor clocked at 2.40 GHz and has 32 GB of RAM. The experimental setup comprises application clients, OPL and data representations. Our baseline setup employs 12 machines in total: 8 as client machines, 1 as an OPL server and 3 as data representation servers.

The dataset comprises 1 million entities, and each entity has a data item related to it at each of the 3 corresponding representations. Unless otherwise mentioned, we

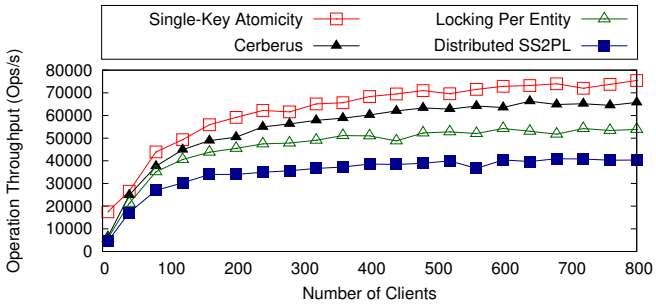


Figure 3.5: Scaling-Up Clients - Uniform Distribution

employ our baseline setup and the workload is uniformly distributed over the entities, and comprises 80% read-only transactions and 20% read-write transactions.

At each representation, data is stored using an *in-memory key-value store*. We develop a simple key-value interface which supports get, put and test-and-set operations. Logging to the disk (Section 3.5) is turned off. Java is used for implementation, and Google's GRPC [90] and protocol buffers [50] for network communication. All the read and write requests are blocking calls. In every experiment, each client issues 10,000 transactions. Each client can issue any transaction type (read-only, read-write, write-only). Measurements were averaged over 3 readings for smoothing any experimental variations.

3.6.4 Experimental Results

Scaling Up Clients

We analyze Cerberus's performance by scaling the number of clients, as illustrated in Figure 4.2. We employ our baseline setup of 12 servers with 1 OPL server, 3 data representations and application clients distributed among 8 servers. The workload comprises 80% read-only transactions and 20% read-write transactions, The workload is uniformly distributed over the entities and the 3 data representations.

The Single-key atomicity scheme achieves the highest throughput. With 800 clients,

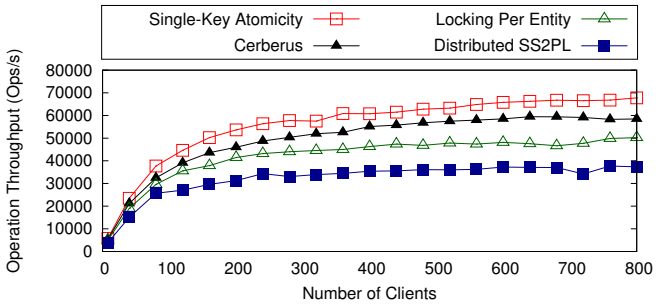


Figure 3.6: Scaling-Up Clients - Uniform Distribution - Varied workload mix

Single-key atomicity scheme achieves a throughput of 75.42K operations/sec and performs 15% better than Cerberus. Cerberus achieves an throughput of 65.75K operations/sec and performs 22% better than the locking per entity scheme and 63% better than Distributed SS2PL. This is because every operation for the Single-key atomicity scheme just atomically reads or writes the data item and no other processing is done to provide consistency guarantees. Hence, although the Single-key atomicity performs well, it can lead to inconsistent semantic executions, like the one described in Section 3.2. Locking schemes perform the worst, but provide the strongest consistency guarantees. Cerberus provides a trade-off in performance and consistency guarantees. Cerberus performs worse than the Single-key atomicity scheme and better than the locking schemes, and still provides implicit causal order guarantees for accessing data across representations.

The gap between Cerberus and the locking schemes increases with the increase in the number of clients. This is because as the number of clients increase, the time spent blocking in the locking schemes adversely affects the performance of the system. Among the locking schemes, locking per entity scheme performs better than Distributed SS2PL. We attribute this to the fact that the locking per entity scheme has to acquire and release less number of locks per transaction, since it acquires locks at the granularity of the entity. Less than 0.05% transactions were aborted for all the schemes.

We also perform the same experiment with a different workload mix, comprising 80% read-only transactions, with 40% of them having a read at a single representation and 40% reading items from two different representations. 10% of the transactions are read-write and 10% of the transactions are write-only. The results are presented in Figure 3.6 and are similar to those observed in Figure 4.2. These results illustrate that even under a workload mix with single item reads and writes mixed with transactions accessing data at multiple representations, Cerberus's performance is close to the Single-key atomicity scheme, while still providing implicit causal order on operations accessing data across representations.

Non Uniform Distribution

Next, we analyze the performance of Cerberus using a non-uniform Zipfian distribution, with zipfian constant (theta) set to .99 (Figure 3.7). We employ the baseline setup of 12 servers. The workload comprises 80% read-only and 20% read-write transactions.

The Single-key atomicity scheme performs the best among the four schemes and Cerberus performs better than the locking schemes. The performance gap between Cerberus and the locking schemes increases as compared to the case with uniform distribution. With 800 clients, Cerberus performs 31% better than the locking per entity scheme and 73% better than Distributed SS2PL. Cerberus provides implicit causal order, which only enforces an ordering among the non-concurrent transactions across the representations, at the granularity of a single entity. This allows Cerberus to better utilize the concurrency for operations accessing the same entity across representations, under higher contention. Furthermore, as Cerberus does not employ locking, it does not suffer from lock contention. These results show that even though Cerberus is optimistic, it performs well under contention and achieves a higher throughput than the pessimistic locking schemes.

For both Cerberus and locking schemes, we see more aborts as compared to the case

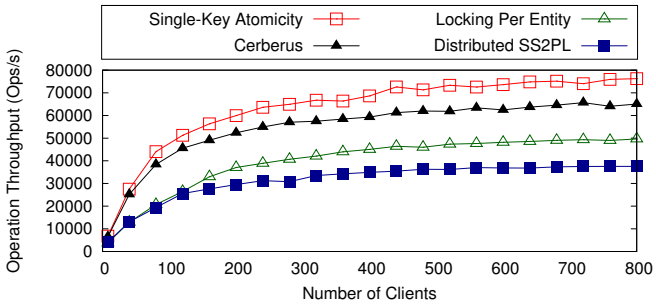


Figure 3.7: Scaling-Up Clients - Zipfian Distribution

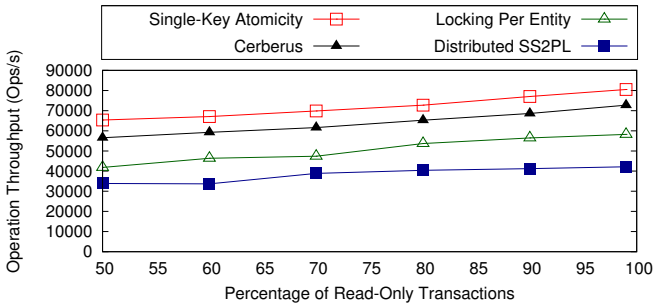


Figure 3.8: Varying Read Write Percentage

with uniform distribution. 7.18% of the transactions are aborted for the locking per entity scheme, 5.4% for Distributed SS2PL and 3.17% for Cerberus. This is due to the contention on the entities accessed more frequently in the workload. Cerberus observes 3 types of aborts: Causal read check violation (amounts for 53% of the aborts), Causal write check violation (46.5%) and Test-and-set failure due to concurrent data access (0.5%). These numbers show that under contention, Cerberus's causal checks help in preventing a significant number of potential consistency violations.

Varying Read-Write Percentage

Figure 4.3 illustrates the performance while varying percentage of read-only transactions from 50 to 99. The number of clients is fixed to 600.

As the percentage of read-only transactions increases, the throughput of all the four

schemes increases, as the reads are less expensive than the writes in the in-memory store. The performance of Cerberus increases at a slightly higher rate. This is because as the percentage of writes decreases, the amount of updates to the RVV (read version vector) decrease as well. RVV is updated during a read of a data item, if there has been an update to another data item related to the same entity, after the recent read of the item. With 50% read-only transactions, around 38% of the reads update the RVV. When the read-only transaction go up-to 99%, then only .8% of the reads lead to updating the RVV. This makes Cerberus well suited to read-heavy workloads like social networks [91].

Scaling OPL

We now study the scaling characteristics of Cerberus in Figure 3.9. The number of OPL servers is varied from 1 to 5, with the number of representation servers being fixed at 3. The entities are range partitioned among the OPL servers equally. Each OPL server processes the transactions which access an entity in the range managed by the server. The number of clients is fixed to 600.

As the number of OPL servers increase, the throughput of all the four schemes increases. Both Single-key atomicity and Cerberus scale better than the locking schemes. With 5 OPL servers, Cerberus achieves a throughput of 165K operations/sec, which is 30% better than the locking per entity scheme and 88% better than the Distributed SS2PL. Scaling the OPL servers provides an advantage to all the schemes, as multiple OPL servers can harness more network bandwidth and processing, and are able to better utilize the cpu at each representation. In Cerberus, OPL only needs to read and update the GVV for the entity accessed by a transaction. Hence, it is suited to scaling OPL servers, as the reading and updating the GVV is now partitioned among multiple servers. The locking schemes still have to acquire locks during each transaction, which restricts the scaling in throughput, as compared to Cerberus. These results show that Cerberus

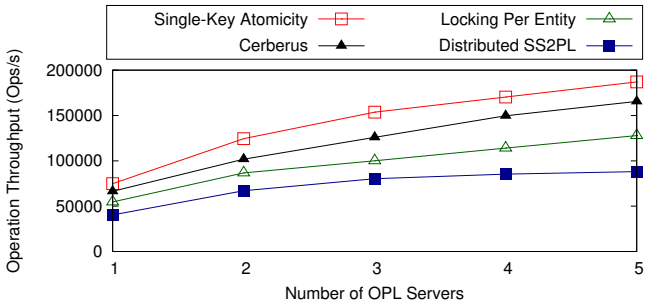


Figure 3.9: Scaling OPL

can efficiently scale via the use of multiple OPL servers.

3.7 Related Work

The proposed multi-representation consistency model, Typhon and and the subsequent developed protocol, Cerberus build on and are related to prior work in the areas of heterogeneous databases, causal consistency and various consistency models in databases.

Heterogeneous Databases. Polystores like Multibase [92], Garlic [93] and BigDAWG [94] aim to store data in heterogeneous engines and provide a query layer over multiple DB engines. However, the updates are local to each data source and the system does not provide any consistency semantics for updates to multiple data sources. Sagas [95] execute transactions over heterogeneous databases by dividing transactions into sub-transactions, and then executing these sub-transactions in a pre-defined order. Although Sagas allow efficient execution of long-lived transactions, they provide weaker guarantees than Cerberus, and can lead to the inconsistent executions discussed earlier in the paper. Altruistic locking [96] provides a mechanism to guarantee globally serializable transactions over heterogeneous database engines. Like Sagas, transactions are divided into sub-transactions. Each sub-transaction can release its locks early at an engine and donate it to other transactions, which then executes in wake of the overall

transaction. The transactions which receive donated locks follow some special rules to ensure a serial schedule. Deuteronomy [97] separates the transaction component and the data component and can used to provide ACID transactions over hybrid data stores.

Breitbart et al. [98] developed the notion of strong serializability to define serializable schedules, which preserve the order of non-concurrent transactions. Typhon's implicit causal dependencies also preserve the order between non-concurrent transactions, but only for transactions accessing the same logical entity, which leads to more permissible histories.

Other weaker correctness criterion than global serializability such as local serializable (LSR) [99, 100], 2LSR [101] have also been proposed for heterogeneous settings. LSR ensures that only local schedules at a database engine are serializable but does not guarantee anything about the global schedule. Some other notions weaker than global serializability, but stronger than LSR, such as 2LSR [101] and QSR [102], have also been proposed. Typhon's consistency model is also weaker than global serializability. It ensures that local schedules are serializable and only imposes orders on global transactions which access the same entity.

Dey et al. [103] develop a technique for providing snapshot isolation guarantees for transactions executing across heterogeneous key-value stores. Reads access a consistent snapshot of the data using snapshot isolation and writes are buffered at the client. Writes are committed using two-phase commit only if the read versions have not changed. Like Cerberus, they rely on the test-and-set feature of the data store to validate that the read versions have not changed. Unlike Cerberus, transactions are not restricted to a set of items (a single entity in Cerberus's case). However, as the read operations only access a consistent snapshot and there is no other coordination mechanism and items across heterogeneous sources are not linked in any way, their protocol can still allow the inconsistent semantic execution scenario we discussed in the paper. Additionally, they

also use synchronized timestamps for versioning, whereas Cerberus uses logical vector clocks, which avoids the need for any time synchronization.

Causal consistency. Lamport [104] introduced the concept of happens-before relationship between events in a distributed system. Ahamad et al. [105] build on this concept and develop a notion of causal memory model, a memory model where reads and write operations respect potential causal dependencies. Many systems provide causal consistency [106, 107, 108, 109, 110] over replicated data. COPS [106] and Eiger [107] provide convergent causal consistency in a geo-replicated setting using explicit dependencies. Each replica only commits an operation after the explicit dependencies have been satisfied. Typhon builds on the notion of potential causality [104] and uses vector clocks [111, 112] to satisfy implicit dependencies at the granularity of an entity in a heterogeneous setting, rather than using explicit dependencies. Cerberus uses both read and update version vectors, due to which it can support read-write transactions, apart from read-only and write-only transactions supported in COPS and Eiger.

Others. Spanner [31] provides external consistency using True-time API in a wide-area setting, ensuring a global order on all operations. However, external consistency needs either atomic clocks, which are expensive or some other method of time synchronization [88, 113]. Deterministic schemes [86, 87] can also be used to provide a global order on all operations. A locking solution like Distributed SS2PL [81] built over all the items at all the representations, preserves the commit order between all non-concurrent transactions. Cerberus enforces implicit causal order on operations at a fine granularity of a single entity and does not need a central sequencer like employed in deterministic schemes or a locking manager, like in SS2PL.

Brantner et al. [114] provide stronger consistency guarantees over key-value stores, for replicated data. A range of consistency guarantees other than causal consistency have also been proposed for supporting single-key get and put access over replicated data [115].

Similar to the single-entity transaction model in Cerberus, some past systems restrict the scope of transactions, based on the application characteristics. In Megastore [40] each transaction can only access a particular fixed key-group, while G-Store [116] provides transactions on flexible key-groups, which are created dynamically by the application.

3.8 Summary

More and more applications have their data spread across multiple database engines, backed by different representations, for supporting diverse data structures, and varied data processing needs. We need to define consistency semantics for such settings, so that we can reason about the guarantees provided by the data management layer and prevent consistency violations for the applications. In this chapter, we presented Typhon, which defines a consistency model for operations accessing application data stored across multiple representations. Typhon uses entities to link data across different representations. Typhon introduces new implicit causal edges in the conflict graph model to capture "happens-before" relationship among operations accessing data across representations.

We also designed a protocol Cerberus, which uses a version vector based technique to provide implicit causal guarantees introduced in Typhon. *Single-entity* transaction model is introduced, which provides the flexibility to support transactions and provide guarantees at the fine granularity of an entity, comprising multiple related items across representations. We formally prove that transactions in Cerberus satisfy implicit causal dependencies. Evaluation results show that Cerberus is scalable, and the performance of Cerberus is close to a solution providing no consistency guarantees across representations and better than locking based solutions.

Part II

Distribution Heterogeneity Within Workloads: Handling Read-heavy and High Contention workloads in OLTP settings

Chapter 4

Scaling Reads in Raft-like Consensus Protocols

4.1 Overview

Many large-scale web applications have to support read-heavy workloads. Social networks have workloads, which are highly skewed towards reads[91, 117, 118]. A study of Facebook's workload [2] reports that the users consume a order of magnitude more content than that being produced, with 99.98% of read requests.

Web applications employ replication to support the large scale of the data and provide fault-tolerance. Data copies are replicated both within [11] and across datacenters [31] to tolerate failures and increase availability of the data being accessed from clients across the globe. However, maintaining consistent replicas to serve user reads across different clusters, or geographically distant datacenters, in the presence of concurrent operations, is a complex task. Consensus protocols are used to provide consistency guarantees about the data, and allow a set of replicas to work together as a coherent group.

Over the last few decades several consensus protocols have been proposed [30, 119, 28].

Raft [28] is a consensus protocol that is designed to be easy to understand and implement. In Raft, writes are replicated to a majority of copies before being committed, and reads are served from a leader node to ensure that the latest value of a data item is read.

Raft's design provides fault-tolerance in presence of failures, but can lead to poor system performance and under utilization for read-heavy workloads. As reads are served only from the leader, read-heavy workloads led to the leader being bottle-necked. Resource under-utilization is exacerbated in failure-free scenarios. Non-leader nodes in Raft end up being cold replicas as long as the leader is active, and are not utilized efficiently.

We propose solutions to scale read-heavy workloads in a consensus protocols like Raft, by utilizing the non-leader nodes, while ensuring that a read returns the latest value of the read. Two approaches: **Quorum reads** and **Strongly Consistent Quorum reads**, are proposed. We also propose a technique to combine the existing leader-based reads with the quorum reads, to configure the system to utilize the cluster uniformly, or tune the system based on read/write latency requirements. We present the proposed approach for Raft, but it can also be applied to Raft-like consensus protocols, such as Multi-Paxos [30].

We integrate the proposed approaches in an existing open-source transactional distributed database, CockroachDB [33], which replicates data, and synchronizes the replicas via Raft. The affect of read-heavy workloads on Raft's performance in CockroachDB is studied, and we illustrate that the proposed approaches help in achieving higher throughput and lower read/write latencies. The implementation of the proposed protocols on CockroachDB has been made available on Github [120].

Next, we first give a background into the system architecture, by describing CockroachDB, and its various components.

4.2 Background: CockroachDB

We employ a distributed database CockroachDB to study the impact of distribution heterogeneity within the workloads. Next, a detailed description of CockroachDB is provided, to give context into the study and the trade-offs involved in the proposed solutions. First, an architectural overview is presented. Then, we describe the two core components of CockroachDB: Consensus layer, which uses Raft, and Transaction layer, which employs timestamp-ordering based protocols.

4.2.1 Architectural Overview of CockroachDB

CockroachDB [33] is an open-source distributed transactional database built on top of a strongly-consistent key-value store. Its primary design goals are scalability and strong consistency. CockroachDB aims to tolerate disk, machine, rack, and datacenter failures with minimal disruption and no manual intervention. CockroachDB provides an SQL based interface to the application clients.

CockroachDB achieves strong consistency by synchronous replication of data. It replicates data over multiple nodes and guarantees consistency between replicas using Raft [28] consensus protocol. Cockroach provides transactional access to data. Transactional access is provided over strongly-consistent data (ensured via Raft). CockroachDB supports two isolation levels: Snapshot Isolation (SI) and Serializable Snapshot Isolation (SSI). Both concurrency-control variations are supported using timestamp-ordering based techniques.

CockroachDB implements a layered architecture as shown in Figure 4.1. The highest level of abstraction is the SQL Layer, which acts as an interface to the application clients. Every SQL statement received at this layer is converted to an equivalent key-value operation. The Transaction Coordinator receives the key-value operations from

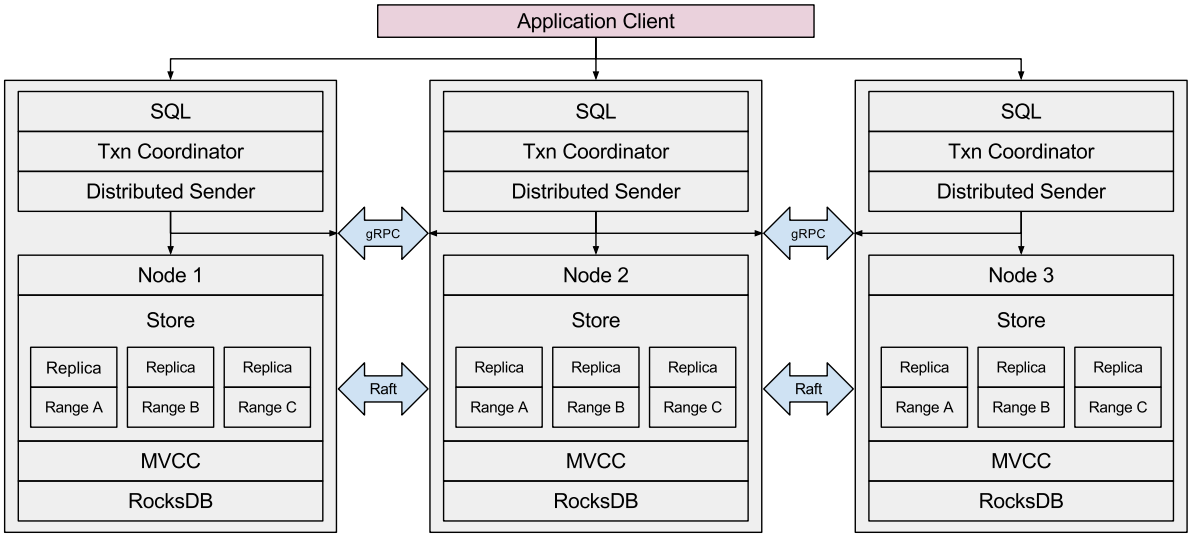


Figure 4.1: Architecture of CockroachDB. The figure shows a cluster of 3 nodes (Node 1 to Node 3), and 3 ranges (Range A to Range C) with 3-way Replication. Every range replicates using its own Raft cluster. Txn Coordinator supports transactional access using timestamp-ordering based concurrency control.

the SQL layer. It creates the context for the transaction, if the corresponding request is the first operation of the transaction. Otherwise, the request is sent to the Distributed Sender in the context of an existing transaction. The Transaction Coordinator also sends a begin transaction request to create a transaction record if the operation is the first write operation of the transaction. Transaction records are used to track the status of the transaction, and accessed for validation by conflicting transactions. The Transaction Coordinator also keeps track of the keys written by the transaction.

The Distributed Sender communicates with any number of Cockroach nodes. Each node contains one or more stores, one per physical storage device in that node. Each store contains potentially many ranges. Each range comprises a contiguous group of keys. A range is equivalent to a partition or a shard. The data in the multiple copies is synchronized using Raft, at the granularity of the range.

The logical entity called Replica is responsible for creating and maintaining transaction records, performing read and write operations by calling the the MVCC (Multi

Version Concurrency Control) Layer and also replicating the transaction records.

The MVCC and the RocksDB layer manage the underlying data, as key-value pairs. Data is stored in RocksDB [121]. RocksDB ensures efficient storage and access of data. Data stored in RocksDB includes the key-value data and all the versions associated to a key, as well as all the consensus state associated with Raft.

CockroachDB has a notion of *write intents*, which is utilized both during concurrency-control and replication. If a write operation is successful (described below), then an intent on the data item is created, indicating an ongoing write. The intents is resolved during the commit phase. If the transaction related to the intent succeeds, then the intent is resolved into a new write.

4.2.2 Consensus in CockroachDB

The Problem of Consensus

Assume a collection of processes that can propose values. A consensus protocol ensures that a single value among the proposed values is chosen. If no value is proposed, then no value should be chosen. If a value has been chosen, then processes should be able to learn the chosen value. Quorums play an important role in consensus algorithms. A quorum is the minimum number of votes that a process has to obtain in order to be allowed to make the decision on behalf of the collection of processes. Depending upon the network access strategies quorums can be as complex as $O(\sqrt{n})$ for grid-based [122], $O(\log n)$ for tree-based [123] or $O(n)$ for arbitrary accesses. Majority quorums assume equal reliability of all members of a cohort. It requires access to $(\lfloor n/2 \rfloor + 1)$ members at any given time for read as well as write operations

Raft

Raft [28] is a consensus protocol that is designed to be easy to understand and implement. It is equivalent to Multi-Paxos [30] in fault-tolerance and performance. It has two phases which are logically separated: **leader election** and **log replication**. A server in a Raft cluster is either a leader, a candidate, or a follower. Once a leader is elected, log replication is done via the leader, using majority quorums.

A leader is elected in a cluster, and is responsible for replicating log to the followers. It also informs the followers of its existence, and is informed about their status, by periodically exchanging heartbeat messages. Therefore, Raft does not need to invoke a leader election for every request. All the writes are coordinated by the leader. The leader ensures that a quorum of nodes, comprising majority of the cluster nodes having the range, replicate the changes before committing the write. Since writes are not guaranteed to propagate to all the members in the cluster immediately (and are performed in two rounds), a strongly-consistent read, which reads the latest value, is performed at the leader. However, a snapshot read can be performed at any of the followers.

Raft in CockroachDB

CockroachDB provides fault-tolerance by replicating data shards or ranges at multiple nodes, and synchronizing the nodes via Raft consensus protocol. In CockroachDB, Raft replicates data at the granularity of a range. Each range comprises a contiguous group of keys, and are defined by start and end keys. They are merged and split to maintain total byte size within a globally configurable min/max size interval. Range sizes default to target 64M in order to facilitate quick splits and merges and to distribute the load. Within a single range, one node (out of the number of nodes replicating the range) is elected leader, and it periodically sends heartbeat messages to the followers.

Leaders and Leases. As CockroachDB uses Raft to replicate ranges, it appoints leaders in the cluster (one for every Raft instance). CockroachDB has an additional abstraction called *leader leases*. Leases are Raft-agnostic and are a sequence of database time intervals which are guaranteed not to overlap, for which a single replica in a Raft group has the leader lease. For this time window, the leader is assumed to be stable, hence avoiding the need to go through the expensive Raft read processing for consistent reads (i.e. waiting to hear from majority using heartbeats to ensure the leadership of the node is valid, after receiving the read request). Reads and writes are generally addressed to the replica holding the lease; if none does, any replica can be addressed, causing it to try to obtain the lease synchronously. The replica holding the lease is in charge of handling range-specific maintenance tasks such as splitting, merging, and re-balancing.

The lease holder can serve consistent reads locally, however, it needs to submit writes to Raft (i.e. go through the leader). The lease is completely separate from Raft leadership, and so without further efforts, Raft leadership and the leader lease might not be held by the same replica. Since it is expensive not to have these two roles co-located (the lease holder has to forward each proposal to the leader, adding costly RPC round-trips), each lease renewal or transfer also attempts to co-locate them.

4.2.3 Transaction Processing and Concurrency-Control

CockroachDB uses a multi-version timestamp ordering protocol to guarantee that its transaction commit history is serializable. The default isolation level is called Serializable Snapshot Isolation(SSI). The SSI mechanism employed in CockroachDB is lock-free and ensures concurrent read-write transactions will result in a seriazable schedule [81].

Each transaction comprises read and write requests, followed by a commit request. Every transaction is assigned a timestamp (by the node on which it starts) when it

begins. This timestamp is used to resolve conflicts with respect to timestamp ordering. As mentioned above, each transaction has a transaction record, that stores the status of the transaction. The transaction record is also replicated via Raft. Every transaction starts with the initial status PENDING. If a transaction is aborted due to data conflicts, the status is changed to ABORTED, or else its status is changed to COMMITTED on commit.

If a transaction is distributed, i.e., the data accessed by the transaction is spread across multiple ranges, then the transaction record is maintained only at one of the servers having data accessed by the transaction. All the operations of the transaction update the same transaction record according to the status of the operation at the particular data server.

CockroachDB keys store multiple timestamped versions of the values. Every new write of a committed transaction creates a new version of the value with a timestamp of that transaction. The write of an uncommitted transaction is added as an intent version with the timestamp of the transaction.

Read operations on a key return the most recent version with a smaller timestamp than the transaction. The timestamp of the transaction performing the read operation is recorded in a node-local timestamp cache. This cache returns the most recent timestamp of a transaction which read the key.

All write operations consult the timestamp cache for the read timestamps of keys they are writing. If the returned timestamp is greater than the transaction's timestamp, then this indicates a timestamp order violation. Hence, the transaction is aborted and restarted with a larger timestamp. Otherwise, the write is successful, and a write intent is created for the item and replicated on a majority of servers via Raft.

On receiving a commit request, the status of the transaction record is updated to commit (if the transaction has not been aborted due to conflicts), and write intents are

resolved to values. As the transaction record and intents are replicated, this operation goes via Raft. If the replication is successful, this ensures that a transaction is committed, the affect of writes will be reflected at a majority of servers.

4.3 Supporting Read-Heavy Workloads

4.3.1 Bottlenecks to read performance

All the read and write requests in CockroachDB are performed via Raft consensus protocol to provide strong consistency, and are submitted to the lease holder. Raft's leader-based design also makes it easy to reason about the correctness of replicas, and simplifies the recovery and configuration change of the cluster. However, this design can lead to under-utilization of resources and can affect system performance, especially for read-heavy workloads.

Irrespective of the number of replicas for a range (or the size of the range) only the lease holder is responsible for serving reads. This could result in the lease holder getting overloaded by requests. This effect can potentially be compounded in the case of hot-spots, where only a few keys are being accessed most of the time. Range splitting (already supported in CockroachDB via MultiRaft [124]) can be used to reduce the effect of hot-spots, but increasing the number of ranges increases the probability of distributed transactions across multiple ranges, which can lead to even worse performance. Read-heavy hot-spots can still occur over a smaller range, leading to poor read performance.

Serving the read requests only at the range-lease holder adversely impacts the system performance as a whole. On the other hand, members other than the lease holder are just cold replicas and are not serving the client directly during failure-free and read-heavy executions. If we could reduce the read traffic on the lease holder by distributing the

requests on other nodes then we can utilize the entire cluster more efficiently and thereby sustain higher throughput.

4.3.2 Designing for Read Scalability

Reducing the load on the lease holder by distributing requests to other nodes can result in the cluster being used more efficiently. Raft uses write quorums before a write can be considered committed. The protocol makes sure that the updates are propagated to a majority of servers in the cluster. This property can be utilized to enable follower nodes to handle read requests. If the reads are performed on a majority of servers in the cluster, it is guaranteed that at least one of the server will have the consistent state. Therefore, read quorums can be used to serve strongly consistent reads from the followers.

We propose two variants of quorum reads to efficiently scale read-heavy workloads. A simple majority read approach, **Quorum read**, provides an efficient solution, but has a pathological case where the value returned might not be the latest linearizable read. This approach provides the advantage of an efficient read, but has a low probability of returning a stale value. The second proposed protocol, **Strongly-consistent Quorum read**, returns the linearizable value corresponding to the key being read, but can be more expensive. Next, we describe both the proposed approaches and how they are executed in CockroachDB.

4.3.3 Proposed Quorum Read variants

Quorum Read

In this approach, the gateway node (the node receiving the client request) sends the read request to a majority of the servers. Every node replies with a timestamp along with the data, corresponding to the latest stable value at the node for the key read.

Since a committed value must be present in one of the servers among any majority, the value corresponding to the highest timestamp is the latest committed value. This value is chosen and sent back to the client.

The pathological case for this approach is that a particular value might be in the process of committing; a majority of servers might have responded to the leader to agree to the write request, but might not have heard back from the leader yet. Hence, the leader might have committed another value, ahead of the value read using the quorum read approach. This approach also provides a trade-off to a local snapshot read, as it is more expensive than reading the local value at a replica, but has a much higher probability of returning the latest value.

Strongly Consistent Quorum Read

To overcome the pathological case of the simple quorum read approach, we also propose a strongly consistent quorum read. As described earlier, Cockroach DB uses write intents for recording the proposed values at a server. When a node receives a request for a strongly consistent quorum read, in the case of write intents for a particular key, the node replies with only the timestamp, but no data. Sending a timestamp and no corresponding data signals that a write intent was encountered for that particular key. As the read is sent to a majority of servers, any possible ongoing write request that could have been committed at the leader will be detected. This is due to the fact that for the leader to commit a value in Raft, a majority must have appended the request to their respective copy of the log. The gateway node then selects the data with the latest timestamp. If the data is available, it implies that the timestamp corresponds to the latest stable value. However, if there is no data available, corresponding to the highest timestamp, it implies that there are pending updates in the system and the request is considered as failed. A back-off policy is used for subsequent retries.

4.3.4 Combining Lease-Holder and Quorum Reads

As write requests still go through the lease-holder, the latest value can be read from the lease-holder like before. We also propose a technique to combine the lease-holder based and quorum reads. If the gateway node is the lease holder, it can retrieve consistent state locally. However other nodes have two choices:

1. Read from the lease holder, or

2. Read from a majority (excluding lease holder)

To read from a majority, a basic approach could be to send requests to all the servers in the cluster (except the lease holder) and wait to hear back from a majority before replying to the client. This approach would work but could end up generating a significant load on all the servers. Instead we can use a random selection approach to send requests to only selected servers in the cluster to form a majority. We make this choice to optimize for failure-free executions.

A read request is executed either as a quorum read or a lease-holder read, such that the read requests are uniformly distributed over all the nodes. Assuming that a cluster of n nodes is fully replicated, every node gets equal number of client requests, and a gateway node always includes itself for majority, the read request to the non-lease holder nodes uses the lease-holder read for $x\%$ of total reads and uses quorum reads for all the other requests. Solving for x for distributing the read requests uniformly in the cluster, we get,

$$x = \frac{P * (n-2)}{n + P * (n-2)} \times 100$$

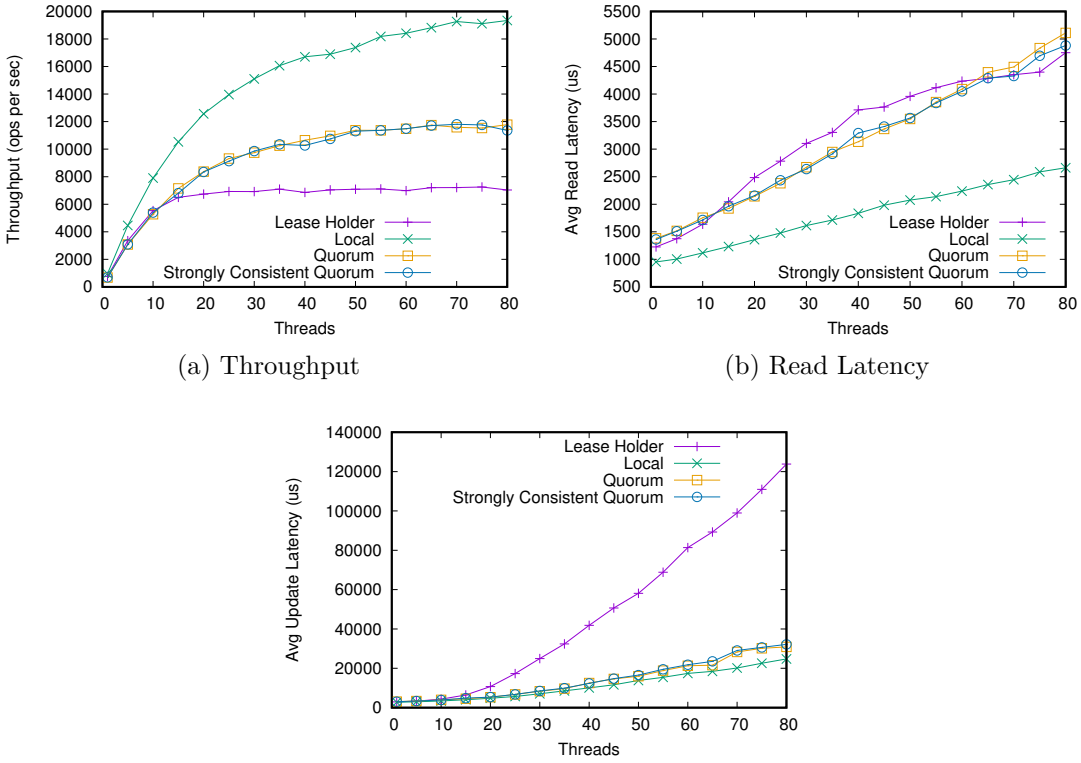
where P is probability of a non lease-holder node being included in a majority quorum by other non-lease holder nodes

$$P = \begin{cases} 1 & n = 3 \\ \frac{\binom{n-3}{\lfloor n/2 \rfloor - 1}}{\binom{n-2}{\lfloor n/2 \rfloor}} & n > 3 \end{cases}$$

Using the combination of lease holder reads and quorum reads we can guarantee consistent reads while not overloading the lease holder. Also, all the cluster members would be utilized for serving read requests and the lease holder would not be a bottleneck. The fraction of reads being served using quorum reads can be configured based on the read and write latency trade-off desired. Having a higher fraction of reads served by the non-lease holder nodes can help reduce the load on the lease-holder and reduce write latencies, at the expense of read latencies and increasing the load of non-leader nodes. On the other hand, a higher percentage of read requests can help in reducing the read latencies, as the read involves only a single node, and does not have to contact a majority of nodes.

4.4 Evaluating Performance of Quorum Reads

YCSB [125, 48] is used to benchmark and analyze the performance of the proposed approach of combining quorum reads with traditional lease holder reads in CockroachDB's Raft implementation. We compare four approaches in the evaluation: *Lease-holder reads* (default baseline approach of reading from the lease holder), *Local reads* (read the local value at the replica), *Quorum reads* (read latest value from a majority), and *Strongly consistent quorum reads* (quorum reads considering ongoing write requests). Local reads may return inconsistent results, but provide a measure of the upper bound of read performance. Both the quorum read and the strongly consistent quorum read approaches also perform a fraction of reads at the lease holder to ensure equal distribution of read



(a) Throughput

(b) Read Latency

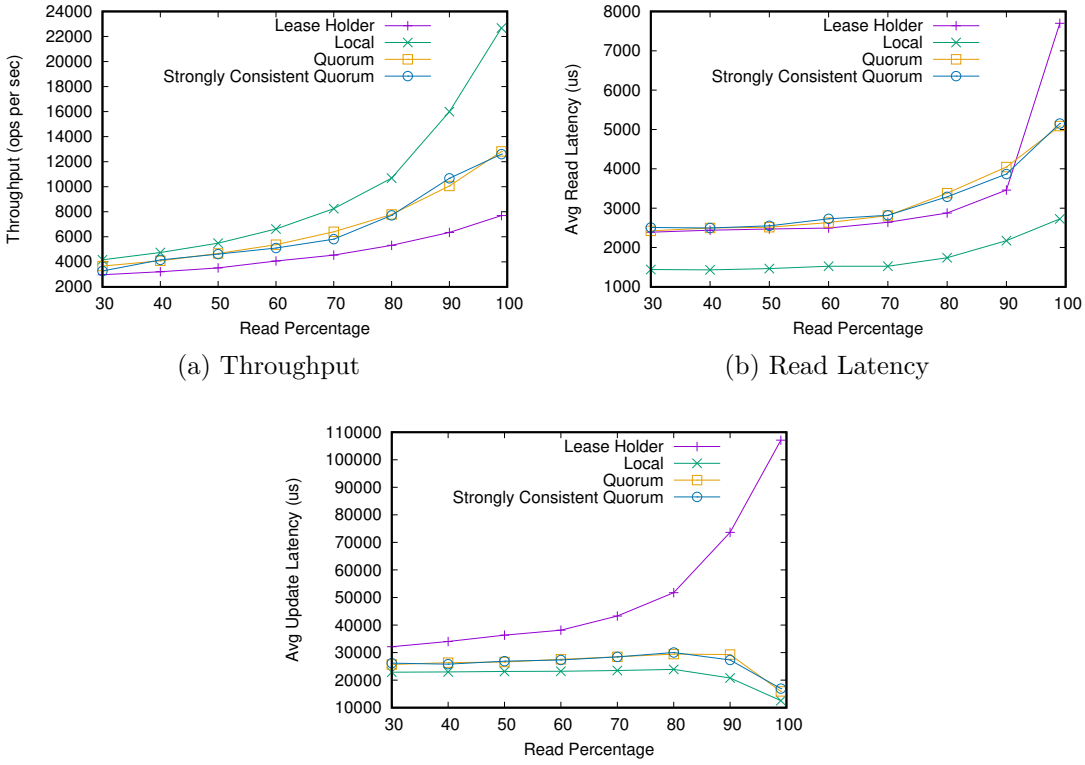
(c) Write Latency

Figure 4.2: Scaling-up clients - Uniform distribution

requests, as described in Section 4.3.4.

We use a fully replicated cluster of 5 AWS EC2 machines (m3.2xlarge instance type), with range size big enough to avoid any range splits during the experiments, so that we can focus on analyzing the impact of the proposed approaches. The fraction of lease-holder reads was set to 28.57%, based on the calculation of x to ensure equal distribution of read requests throughout the cluster (Section 4.3.4). Another machine was used as YCSB client to generate equal load on all the nodes. YCSB generates single-key read and write operations. These are send to CockroachDB as single operation transactions. For processing of write operations at the transactional layer, the steps for write and commit operations, described in Section 4.2.3, are combined.

The dataset comprises 100K records with each record having a key and a value. Unless



(a) Throughput

(b) Read Latency

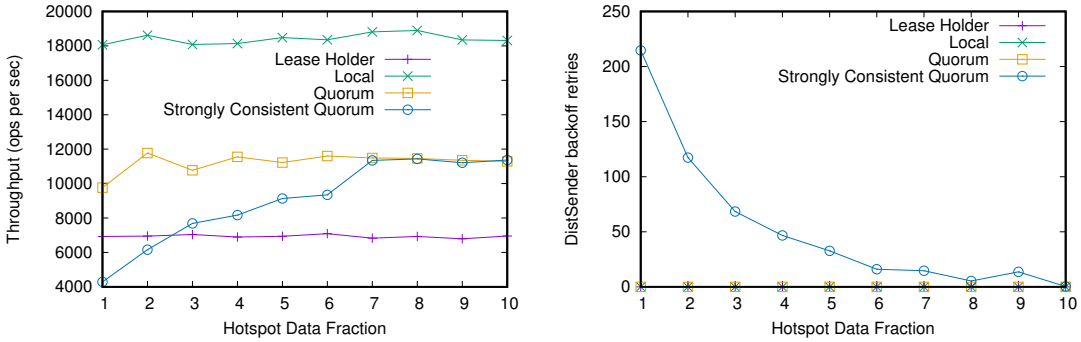
(c) Write Latency

Figure 4.3: Varying percentage of reads - 70 client threads

otherwise mentioned, the workload is read-heavy, with 95% reads and 5% writes, and is uniformly distributed over the keys. The servers were monitored using nmon [126] to track resource utilization.

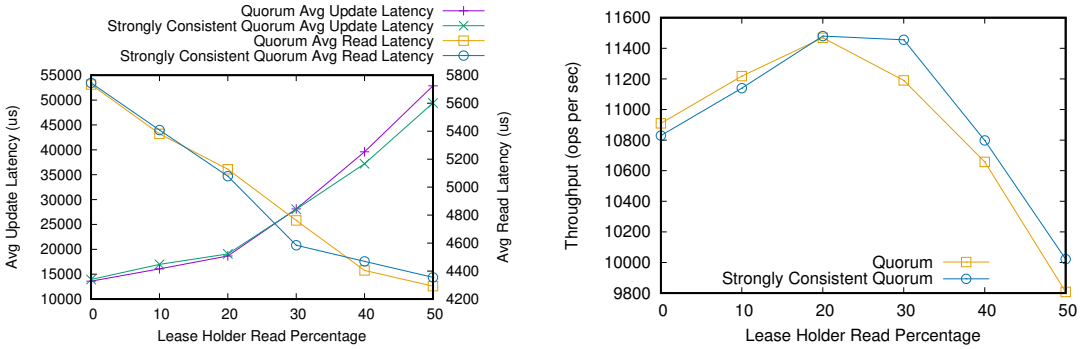
Scaling-up Clients

Figure 4.2 illustrates the performance of all the approaches under varying number of client threads. We observe that both the quorum read and strongly consistent quorum read achieve higher throughout than the traditional lease holder read approach (around 60% higher), where all the reads are performed at the lease holder. Figure 4.2c also shows that distributing the reads is highly beneficial in mitigating the increase in write latency, as the load on the system increases. As both the quorum read approaches reduce the



(a) Throughput (b) Distributed Sender Retries

Figure 4.4: Varying hotspot data fraction - 70 client threads



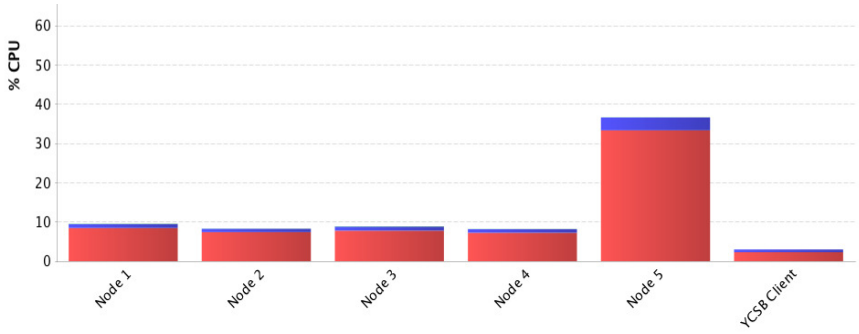
(a) Read & Write Latency (b) Throughput

Figure 4.5: Varying fraction of lease-holder reads - 70 client threads

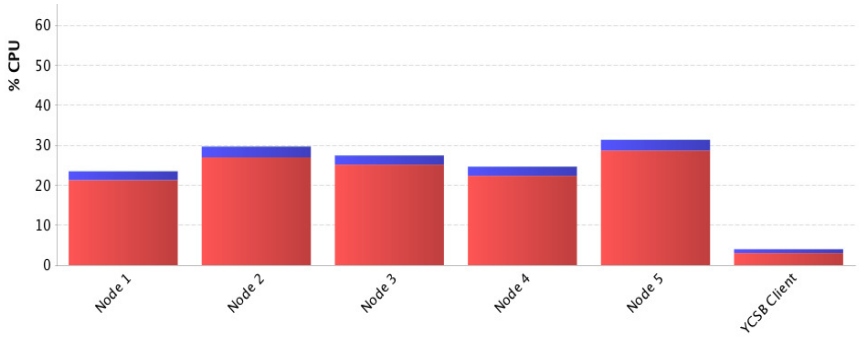
load on the lease holder, around 4x improvement in write latency is observed with 80 concurrent client threads. Both the proposed approaches achieve a lower read latency with more than 10 client threads, due to the distribution of read requests. But as the load on the system increases, the higher number of reads involved in the quorum reads result in reducing the gap in read latencies.

Varying Read-Write Ratio

In Figure 4.3, we observe the effects of varying percentage of reads from 30 to 99. As the percentage increases, the throughput gap between the quorum read approaches



(a) Lease-holder reads



(b) Strongly consistent quorum reads

Figure 4.6: Average CPU utilization - 70 client threads

and the traditional lease-holder read approach also increases (Figure 4.3a). This is due to the fact that increasing fraction of reads can be efficiently distributed among the cluster via quorum reads, resulting in better throughput. Figure 4.3b shows that upto a certain fraction of read requests, lease-holder reads achieve a better read latency. This is because of the lease-holder read only accesses a single node, and does not have to contact a majority of servers. However, after a certain percentage of read requests, read latency for lease-holder reads increases at a much faster rate when compared to quorum read approaches. Again this can be attributed to the fact that a very fraction of lease-holder reads overload the lease-holder. Additionally, Figure 4.3c shows that offloading the lease-holder using quorum read approaches helps in drastically reducing the write latency for read-heavy workloads.

HotSpots

Figure 4.4 illustrates the performance under hotspots, where 80% of the requests access a restricted fraction of the data (specified in % of the entire data on the horizontal axis). We observe that under really high contention (left side of the horizontal axis), the throughput achieved by the strongly consistent quorum read approach is much lower than quorum reads (Figure 4.4a), because of the higher probability of accessing a key within an ongoing write request. While, as the contention decreases, the throughput gap between the two proposed quorum read approaches decreases. For other approaches, the throughout does not vary a lot due to the low percentage of write requests. Figure 4.4b shows that the number of retries at Distributed Sender (or gateway node) increases exponentially as the hostspot data fraction decreases. Again this is due to the fact that probability of encountering a write intent increases as the contention increases.

Read-Write Latency Trade-Off

In the proposed quorum read approaches, the fraction of reads to be performed at the lease-holder provides a way to trade-off read and write latencies. Figure 4.5a illustrates this trade-off. As we increase the fraction of reads performed at the lease holder, the write latency increases due to the increase in load on the lease holder. On the other hand, we also observe that serving more reads from the lease holder at a high load, reduces the read latency. As seen in Figure 4.5b the highest throughput was achieved at 20% lease holder reads, when the load is almost uniformly distributed throughout the cluster. Combining the quorum read approaches with the traditional lease holder reads, provides a mechanism to trade-off between read and write latencies.

Resource Utilization

Figure 4.6 shows the average CPU usage for all the machines (including 5 CockroachDB nodes and 1 YCSB client) during different read approaches. Node 5 is the lease-holder in both the figures. In case of lease-holder reads (Figure 4.6a), the lease-holder node has comparitively higher CPU usage than other nodes. However in case of strongly consistent quorum reads (Figure 4.6b), all the nodes in the cluster are almost equally used. Therefore by using idle resources, quorum approaches provide better throughput.

4.5 Related Work

El Abbadi et al. [127, 128] introduce the idea of views and Viewstamp replication [119] builds on the work, combining views with the concept of a leader (or the primary) in a given set of members (called the cohort), for replicating a state machine and recovering on failures. Paxos [29] is a general-purpose consensus protocol designed for an asynchronous environment. In Paxos, each process plays the role of a proposer, an acceptor, or a learner. Every request requires a new instance of Paxos in the system. Each request has to go through two phases to get accepted: proposal phase and acceptance phase. Consistent reads are performed using quorums (because of no stable leader). Multi-paxos [30] removes the need for electing a leader (the proposal phase in Classic Paxos) for every single request to reduce the number of rounds. Modifications like Master-leases [129] in Multi-Paxos enable serving consistent reads locally at the leader.

Quorum leases [130] allow any replica to acquire a lease from a majority of grantors and serve consistent reads locally. This is made possible by synchronously notifying every write to all the lease holders through the lease grantors. Other read optimizations include snapshot reads using synchronized clocks in Spanner [31] and local reads in Mega-

store [40]. Zab [131] is a crash-recovery atomic broadcast algorithm that was designed for ZooKeeper [132] coordination service. It is similar to Paxos in some key aspects except that it requires stricter ordering guarantees due to incremental state changes.

Many large scale web-applications also employ caching to scale reads [117]. Such approaches are orthogonal to our approach, and can be used in conjuction with the proposed approaches.

4.6 Summary

Read-heavy workloads are widely prevalent in many web applications. We propose combining quorum reads, with traditional single leader based reads in Raft-like consensus protocols. A basic quorum read approach, which can return non-linearizable value in a corner case and a strongly consistent quorum read approach are proposed. We implement our approach in open-source distributed database, CockroachDB, which employs Raft for consensus. Results with YCSB benchmark demonstrate that the proposed approach can result in higher throughput and improved read/write latencies with read-heavy workloads, and can result in better utilization of the follower nodes in read-heavy and failure-free scenarios.

Chapter 5

Dynamic Timestamp Allocation for Tackling High Contention Workloads

5.1 Overview

Many OLTP workloads have high data access skew, where a small portion of data objects receive a high percentage of overall requests. Social networks like Facebook have some lynch-pin objects (Celebrities, Product pages etc), which have an order of magnitude higher rate of access, as compared to other application objects [133]. Similar characteristics are also observed for financial applications. Hot-Spots or high contention on some objects can also be caused by breaking news events [6], or cyclic events like Holiday Season.

The high contention in OLTP workloads can have a big impact on performance of applications. Hot-spots can lead to a significant drop in transactional throughput. Figure 5.1 illustrates the performance of the distributed database, CockroachDB (which is described in detail in the last Chapter in Section 4.2), with increasing contention in the transaction workload. As contention increases, the number of aborts increases, leading

to a drop in throughput.

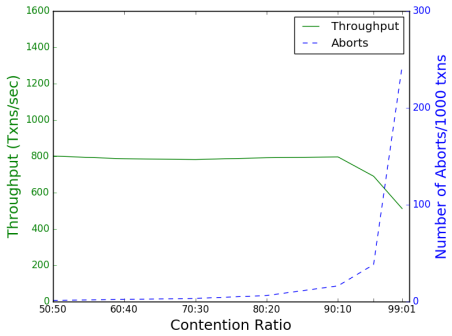


Figure 5.1: Performance of OLTP under Contention in CockroachDB. Contention ratio x:y specifies that x percent of the transactions access y percent of the items.

As described in Chapter 4.2, CockroachDB employs a Serializable Snapshot Isolation (SSI) based approach for concurrency-control. To implement SSI, CockroachDB employs a lock-free multi-version timestamp ordering scheme, which is based on the classic pessimistic timestamp ordering technique. The timestamp ordering scheme in CockroachDB uses a fixed timestamp allocation scheme, and assigns timestamps at the start of each transaction. These timestamps are used as commit timestamp for the transaction. This commit timestamp is used to order the transactions in logical timestamp order, and hence enforce serializability. This ensures serializability, and provides a lock-free technique for concurrency control, which is useful in a distributed setting. The affect of blocking in locking schemes, can be exacerbated under node or message failures in a distributed environment. The restrictive fixed timestamp ordering, however, leads to higher number of aborts under contention.

To reduce the number of aborts at high contention, a possible strategy is to use dynamic timestamps for allocating the commit timestamp to a transaction. In this technique, the system tries to dynamically allocate a timestamp range to each transaction, based on the conflicts of data items accessed by the transaction. At commit, a timestamp is allocated to the transaction from that range, to fit that transaction on a logical

serializable timeline. We integrate CockroachDB with the dynamic timestamp allocation technique we designed in our previous work MaaT (Multiaccess as a Transaction) [37]. MaaT changes the validation phase of the optimistic concurrency control mechanism to allocate a commit timestamp dynamically by keeping track of the items accessed and conflicting transactions accessing the items using read and write markers. By performing dynamic timestamp allocation during validation and commit, MaaT is able to reduce aborts under contention.

Past techniques proposed to reduce aborts in lock-free concurrency control techniques [134, 37, 135] have been implemented and evaluated in standalone prototypes. Implementing and analyzing the performance of dynamic timestamp ordering technique in a full-featured database system like CockroachDB gives an insight into the performance characteristics of the dynamic timestamp ordering like optimizations, and how to integrate such optimizations in existing systems.

A benchmark is also developed to extensively evaluate the performance of dynamic timestamp ordering based concurrency-control in CockroachDB. The benchmark evaluates the performance while varying contention, the degree of concurrent access and the ratio of read-only and read-write transactions. The source-code changes as well as the benchmark are available on Github [136, 137].

CockroachDB's architecture and transaction processing mechanism has already been described in detail in Section 4.2.1 and Section 4.2.3, respectively. Next, we give an overview of MaaT, and then describe how MaaT is adapted and integrated within the concurrency control mechanism in CockroachDB.

5.2 MaaT overview

MaaT (Multi-access as a Transaction) [37] re-designs the optimistic concurrency control (OCC) protocol in order to make it practical for distributed, high-throughput transactional processing systems. To achieve high-throughput with update intensive workloads, MaaT assigns dynamic timestamps to the transaction, instead of fixed timestamps, and re-designs the verification phase of OCC so as to reduce the transaction abort rate. In particular, a mere conflict between two transactions should not be enough to restart them; instead, the system should try to figure out whether this conflict really violates serializability, and should tolerate conflicts whenever possible.

While performing read and write operations, each transaction maintains and updates valid timestamp ranges it can commit in, based on the data items accessed and conflicting operations on those data items. At commit time, if there is a valid timestamp range for the transaction to commit, then a commit timestamp will be chosen from that valid range. Employing timestamp ranges allows the flexibility to shift the commit timestamp to fit it in the logical serializable order.

Design

MaaT assigns a timestamp range with a lower and upper bound for each transaction, instead of a fixed timestamp. Initially, the lower bound is set to 0 and the upper bound is set to ∞. The lower and upper bounds are adjusted dynamically with respect to the data conflicts of the transaction. MaaT employs soft locks, which do not block transactions, to act as markers to inform transactions accessing a data item about other transactions that have read or written that data item but not committed yet.

Each transaction can either be single-sited or distributed. For a distributed transaction, one of the servers involved in the transaction is declared the transaction coordinator.

Timestamp range is maintained at each of the servers accessed by the transaction. Next, we describe how the timestamp range are dynamically adjusted, and used to allocate commit timestamp to a transaction. This is one of the core components of MaaT, which leads to reduced aborts under contention.

If T and T' are two transactions, with lower bounds, *lowerbound(T)* and *lowerbound*(T') and upper bounds, *upperbound*(T) and *upperbound*(T'). Whenever T reads a data item, which has a write soft lock by T', MaaT adjusts the *upperbound*(T) to be less than *lowerbound*(T'), so that T executes as if did not see the updates made by T'.

Whenever T writes a data item, which has a read soft lock by T', MaaT adjusts the *lowerbound*(T) to be greater than *upperbound(T')*, so that T' executes as if it did not see the updates that will be made by T.

Whenever T writes a data item, which has a write soft lock by T', MaaT adjusts the *lowerbound*(T) to be greater than *upperbound(T')*, so that T' executes as if it did not see the updates that will be made by T.

In the validation phase, the timestamp range of T and/or the timestamp ranges of other transactions are adjusted to ensure that the timestamp ranges of conflicting transactions do not overlap. The outcome of these validation operations is to determine whether the constraints on the commit timestamp of T can be satisfied or not; that is, whether T can commit or not, by finding a timestamp to fit T in the logical serialization order. Transaction T is aborted if there is an overlap of the timestamp range with other concurrent transactions or if the lower bound of T is greater than its upper bound; otherwise, T is committed, and the client picks an arbitrary timestamp from the intersection range to be the commit timestamp of T.

If T is a distributed transaction, each data server accessed by T adjusts the timestamp range on the server based on the items accessed there. All these ranges are sent to the coordinator of the distributed transaction. If there is a valid intersecting range in the

timestamp ranges sent by all the servers, the transaction is committed. Otherwise, the transaction aborts.

5.3 Abstract Overview of Dynamic Timestamp Ordering Adaptation in Cockroach DB

Employing SSI in CockroachDB ensures correctness, but it decreases the performance at high contention. Adapting SSI approach in CockroachDB to work with dynamic timestamps will avoid aborting transactions that can be serialized by changing their timestamps with regards to their data conflicts. Next, an example is analyzed to illustrate how dynamic timestamping can lead to reduced aborts as compared to the current concurrency control technique employed in CockorachDB. We then describe the data structures introduced in CockroachDB to enable dynamic timestamping mechanism.

5.3.1 Reducing aborts with dynamic timestamping

When the isolation level in CockroachDB is set to SSI, transactions are aborted in following cases:

1. Conflicts are analyzed at every write, to check whether any later transactions have read or written the data item currently being written by the transaction. If this is the case, the transaction is aborted.

2. When reading an item, if there is a write intent created with a lower timestamp, then the SSI approach in CockroachDB will abort one of the transactions. The transaction to abort will be decided based on the priority of the transactions involved.

Dynamic timestamp allocation can help reduce some of the aborts in the above cases. It allows more concurrent operations to commit by dynamically trying to commit the transaction on a logical timeline based on order of access of items. Consider the following example.

Example 1 *Lets consider the following two transactions.*

$$T_1 : r_1(y)\ r_1(x)$$
$$T_2 : r_2(x)\ w_2(x)$$

Suppose the execution history comprising the two transactions is as follows.

$H_1 : b_2\ r_2(x)\ b_1\ r_1(y)\ r_1(x)\ c_1\ w_2(x)\ c_2$

Transaction T_2 begins before T_1, and hence is assigned an earlier timestamp than T_1. At $w_2(x)$, the SSI approach in Cockroach DB sees that x is read by T_1, which is a transaction with a later timestamp. To ensure timestamp ordering given by the fixed timestamp allocation, transaction T_2 is aborted since it causes RW conflict with T_1, and violates the logical timestamp order according to the allocated timestamps.

In case of using the dynamic timestamping technique, suppose $lowerbound(T_1)$ and $lowerbound(T_2)$ are the lower bounds and $upperbound(T_1)$ and $upperbound(T_2)$ are the upper bounds of the transactions T_1 and T_2 respectively. On detecting the RW conflict at w_2, the $lowerbound(T_2)$ is made greater than $upperbound(T_1)$ so that history will be equivalent to the serialization order $T_1 \longrightarrow T_2$. Rather than aborting the transaction due to an initially allocated timestamp order, the dynamic timestamp allocation can re-order the logical transaction order, and lead to commitment of both the transactions in this case.

5.3.2 Data Structure changes in CockroachDB

Transaction Record

CockroachDB maintains a transaction record for each transaction, which maintains the transaction ID and transaction state. We add two fields to hold the lower and upper bounds of the transaction. *CommitBeforeQueue* field is added to hold the list of transactions, which have the condition that the current transaction has to be serialized before them. *CommitAfterQueue* holds the list of transactions after which the current transaction can commit.

Timestamp cache

For each data item CockroachDB maintains the timestamp of last read and last written transaction in its *timestamp cache*. In the SSI approach, this can be updated even by uncommitted transactions. We modify the mechanism updating the timestamp cache in CockroachDB to ensure timestamps in the read cache are updated only by committed transactions.

Soft Locks and Soft Lock Cache

Soft Locks are non blocking markers to inform other transactions about ongoing transactions. Soft Locks comprise transaction-metadata, which is used to locate the transaction record of the transaction that placed the soft lock. A read soft lock is placed while reading and a write soft lock is placed while writing the data items. The Soft Lock Cache holds the read soft locks and write soft locks per key.

5.4 Integrating dynamic timestamp ordering in CockroachDB

We now describe the integration of dynamic timestamping mechanism in CockroachDB. First, handling of different operations of the transaction is described. Then, we describe the changes implemented at different layers in CockroachDB, for integrating the dynamic timestamp ordering based concurrency-control mechanism.

5.4.1 Transaction Lifecyle

As described earlier, a transaction comprises of begin, read, write and commit operations. On every read operation, a soft read lock is placed on the key being read and soft write locks that are already placed on the key are collected. The soft write locks are then placed in the corresponding transaction record. The *CommitBeforeQueue* is populated with the transactions which correspond to the soft write locks on the data item, indicating that the transaction reading the item should commit before all the transactions, which have a soft write lock on the item. As the read of the item does not reflect the update of the transactions intending to write the item, its logical commit order should be before such transactions. *CommitBeforeQueue* is used during transaction validation to enforce the transaction ordering implied by the queue.

On every write, instead of aborting transactions on detecting a conflict based on fixed timestamps, a soft write lock is placed on the key being written and, read and write soft locks that are already placed on the key are collected and placed in the transaction record. The *CommitAfterQueue* is populated with the transactions which correspond to the soft read locks on the data item, indicating that the current transaction intending to write the data item, should commit after the transactions which have a soft read

lock on the item (implying the intention to read the data item). As the read performed by the transactions placing the soft read lock, does not reflect the write operation of the current transaction, the current transaction's logical commit order should be after the transactions with the soft read lock. *CommitAfterQueue* is also populated with the transactions which correspond to the soft write locks on the data item, indicating that the current transaction intending to write the data item, should commit after the transactions which have a soft write lock on the item.

Along with the above described processing, on every read or write operation, the lower bound of the transaction is adjusted to be equal to the last committed write timestamp (or read timestamp), which is retrieved from the timestamp cache.

When a commit request for the transaction is sent, a validation phase is executed. During the validation phase, the lower and upper bounds of the transaction are adjusted such that the transaction commits before all the entries in *CommitBeforeQueue* and commits after all the entries in *CommitAfterQueue*. The *CommitBeforeQueue* and *CommitAfterQueue* have been populated with all the transactions, which have conflicting operations with the given transactions. Hence, respecting the commitment order enforced by the queues, guarantees that the transaction orders all conflicting operations and preserves serializability. The upper bound of the transaction is updated to be the minimum of its current upper bound and the lower bound of each transaction in the *CommitBeforeQueue*. Similarly, the lower bound of the transaction is updated to be the maximum of current lower bound and the upper bound of each transaction in the *CommitAfterQueue*.

The timestamp range of the transaction is then checked to see if the lower bound of the transaction is less than its upper bound. If so, the transaction is committed by picking the lower bound, as the commit timestamp; otherwise the transaction is aborted. The transaction status in the transaction record is updated accordingly as COMMITTED

or ABORTED.

On both commit and abort of the transaction, soft read locks and soft write locks held by that transaction are released. On commit, the write soft locks will be resolved to actual write operations in the DB. The read and write timestamps of the items accessed by the committed transaction will be updated in the timestamp cache.

If the transaction spans across multiple ranges, RPCs are used to update the transaction record during read and write operation and to validate the transaction at the remote range (employing the validation strategy to update transaction bounds described above) during validation phase. The soft locks in the remote ranges are resolved asynchronously using RPCs, while the soft locks local to the range are resolved synchronously during the end transaction request.

The dynamic timestamp allocation based proposed design makes use of the soft locks to detect the conflicting transactions. It then tries to reorder the transactions by analyzing these conflicts and allocating dynamic timestamps to the transactions to fit them in a logical serializable timestamp order.

5.4.2 Implementation Details

Multiple components have been modified in CockroachDB to integrate the dynamic timestamp ordering approach.

The Transaction Coordinator is modified to send the begin transaction request to create transaction record on the first operation of the transaction rather than on first write of the transaction. Transaction coordinator now additionally tracks the keys read by the transaction along with the keys written by the transaction.

Transaction record is modified to hold the lower bound and the upper bound of the transaction, with zero and infinity being the initial values respectively. Two new queues,

namely *CommitBeforeQueue* and *CommitAfterQueue*, are introduced in the transaction record. These queues are used at validation to adjust the lower and upper bounds of the transaction based on conflicts of the transaction.

At each replica, the timestamp cache is modified to hold only the timestamps of the committed transactions. A replica is also responsible for maintaining the soft lock cache for all the keys in that replica.

APIs for performing writes at the MVCC layer (such as MVCCPut, MVCCInitPut etc.) are modified to not place the intent, and instead place soft write locks against the key intended to write. Read operation APIs in MVCC (such as MVCCGet, MVCCScan etc.) are modified to place soft read lock on the key being read along with reading the value for the key. New MVCC APIs are created to resolve write soft lock on commit, and write the committed values to the key-value store and to garbage collect soft locks on abort.

New RPCs are created to update the transaction record at the remote range for distributed transactions. Additionally, these RPCs are also used to perform validation on the transaction record in the remote range. Additional RPCs are created to execute commit or abort processing after the validation phase and to resolve remote soft locks asynchronously on commit. Another RPC is introduced to garbage collect remote soft locks asynchronously on aborting a transaction.

The Store layer is modified to handle the asynchronous resolving and garbage collecting of remote soft locks, like the asynchronous resolving of write intents that was done before, for the pessimistic timestamp allocation mechanism.

The source-code changes are available on Github [136].

5.5 Evaluation

CockroachDB is extensively evaluated to compare the performance of the fixed timestamp allocation scheme, with the dynamic timestamp ordering scheme, under varying levels of concurrent access, contention and read-write ratios.

5.5.1 Experimental Setup

Benchmark Description

A YCSB-like [48] benchmark is designed for performing the evaluation. The benchmark provides support for transactions, rather than only key-value operations, like in YCSB. Every transaction generated by the benchmark can be a read-only transaction or a read-write transaction, with both read and write operations. Every transaction consists of 5 operations. The benchmark has 3 parameters that can be altered to test different scenarios. The configurable parameters for the benchmark are described below.

- **Concurrent transactions.** This parameter is used to define the number of transactions that occur concurrently, with a default value of 50. If the *concurrent transactions* is set to 50, the benchmark creates 50 threads and each thread runs a transaction sequentially i.e., if a transaction is performing 3 reads and 2 writes, each of the operations is blocking and the thread executes these operations one after the other.

- **Contention ratio.** To define and vary data access skew in our experiments, we employ the parameter, contention ratio, which indicates skew on a subset of data. The contention ratio 70:30 indicates 70% of the data items will be accessed by 30% of the transactions, while the remaining 30% of the data items will be accessed

by remaining 70% of the transactions. The default contention ratio is a uniform distribution of 50:50, which corresponds to the lowest contention of 50%.

- **Read-only ratio.** In order to see how the dynamic timestamp ordering works for various scenarios such as write intensive transactions or read-dominant transactions, we introduced the read-only ratio parameter. Read-only ratio defines the ratio or the percent of transactions that perform only read operations. For the default value 50:50, 50% of the transactions have only read operations (5 per transaction) and rest of the transactions will be read-write transactions with 3 read operations and 2 write operations. Before beginning a new transaction, we toss a coin with the defined bias based on the read-only ratio parameter value and decide whether the transaction is going to be read-only or read-write.

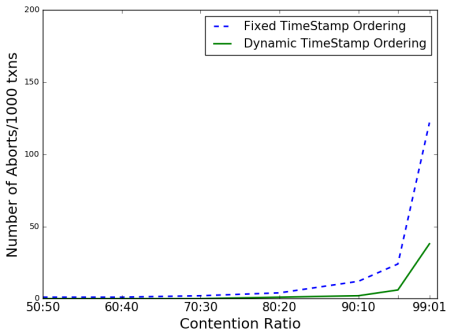
Experimental Configuration

Every experiment was performed on a cluster with 3 servers. Each machine in the cluster runs a 8-core Intel Xeon E5620 processor clocked at 3.8 GHz and has 16 GB of RAM. Each server had one instance of CockroachDB node running on it and the data was replicated three ways. The experiments were run with 100,000 key-value data items without data partition, hence the experiments did not have distributed transactions. Each data point in the benchmarking process corresponds to an experiment performed with 100,000 transactions accessing 100,000 data items. 10000 warm-up reads are performed before starting each experimental run. Each data point reported in the results is an average of 3 repetitions of such an experiment. The benchmark is available on Github [137].

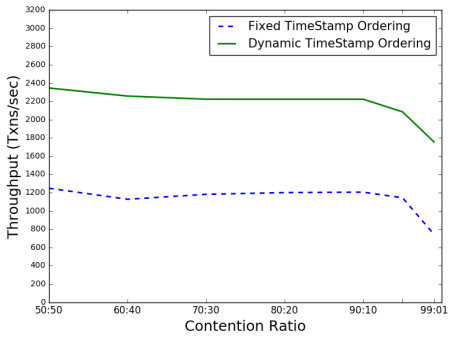
Experimental Methodology

As mentioned in 5.3.2, for dynamic timestamp ordering, the creation of a transaction record happens on the first operation, be it a read or a write, rather than on the first write operation, as is the case with fixed timestamp ordering. In fixed timestamp ordering, if the transaction consisted of all read-only operations, no transaction record is created. Furthermore, the creation of the transaction record also encompasses the replication of the transaction record (to the replicas holding the range associated to the first key accessed by the transaction) using Raft, both at the begin and when the commit decision is made. Hence using this approach in a read-dominant system, adds significant delays compared to the original implementation of CockroachDB. There are many approaches to avoid the bookkeeping done by MaaT for read-only transactions. One such solution was proposed by Agrawal et al. [138], which can be implemented in CockroachDB along with MaaT. A read-frontier based on this approach proposed by Agrawal et al. [138] can be continuously maintained, which provides access to the latest commit timestamp below which no transaction can commit. The timestamp corresponding to the read frontier can be used as the timestamp to read the items accessed in the read-only transactions.

In order to apply optimized solutions for read-only transactions, we first need to identify if the transaction will be read-only. After a discussion with CockroachDB developers, we learnt that there is no provision as of now in the database to specify a transaction as read-only. Implementing this change will require modification in multiple layers of the database and is a complex task, and it is not in the scope of this paper. In order to overcome this issue, in our experiments, we assume that every read-only transaction is successful and will not have the overhead of creating and maintaining a transaction record. If the optimization was to be implemented, read-only transactions would hit only one server, which would respond to the client without creating and replicating transac-

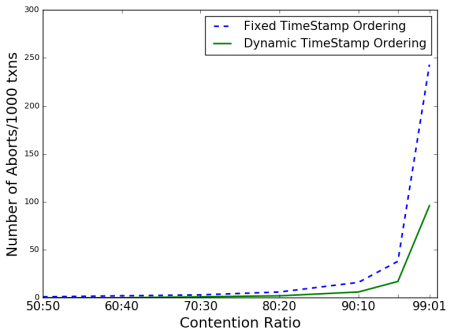


(a) Aborts

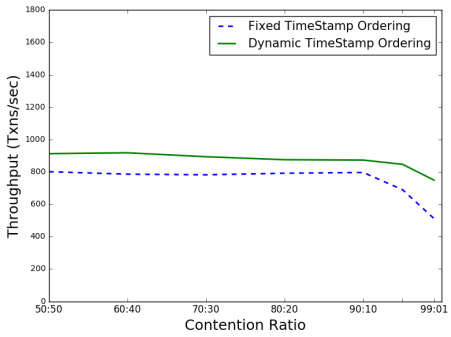


(b) Throughput

Figure 5.2: Varying contention with 80% read-only transactions



(a) Aborts



(b) Throughput

Figure 5.3: Varying contention with 50% read-only transactions

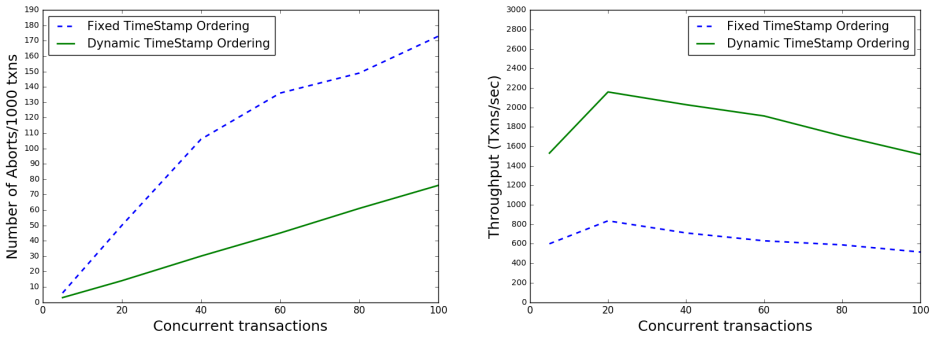
tion record among the replicas of the queried data. So we do not account for the time taken to process and perform read-only transactions. Because of the above mentioned assumption, the throughput displayed in the following sections will be an upper bound of the actual throughput for dynamic timestamp ordering. But the number of aborts or retries is accurate because if the read-only optimization is implemented, it does not cause any read-only transaction to fail; all the aborts will be due to read-write transactions which is captured in our experiments.

5.5.2 Evaluation Results

Varying Contention

First, we analyze the performance of the concurrency-control schemes under varying levels of contention on the data items. We performed two experiments with varying contention from 50% to 99%. In one, we set the read-only ratio and concurrency level with the default values of 80:20 and 50 respectively. Figure 5.2a plots the number of aborts for every 1000 transactions with increasing contention for both dynamic and static timestamp ordering. Even for 20% write transactions, when the contention is at its highest, the fixed timestamp ordering scheme has roughly 3 times more aborts for every 1000 transactions, as compared to dynamic timestamping. Figure 5.2b compares the throughput of both the techniques and we observe that dynamic timestamp ordering has significantly higher throughput. In the second experiment with varied contention, the read-only ratio was lowered to 50:50, while keeping the concurrency level to 50. Figures 5.3a and 5.3b illustrate the results. As mentioned in the methodology, the throughput for dynamic timestamp ordering is an upper bound, especially for higher read-only ratios.

In both Figures 5.2a and 5.3a, although both the techniques started with small abort numbers on low contention, dynamic timestamp ordering results in significantly lower number of aborts with the increase in contention. When contention is high, say 99:01, 99% of the transactions are trying to access 1% of data. Dynamic timestamp ordering ends up accomodating more transactions for commitment due to its flexibility in shifting the lowerbound and upperbound of commit timestamps for all contending transactions. In case of statically assigned timestamps, conflicts arise since large number of transactions are competing to access same set of data simultaneously and the timestamps are fixed, leading to higher aborts.



(a) Aborts (b) Throughput

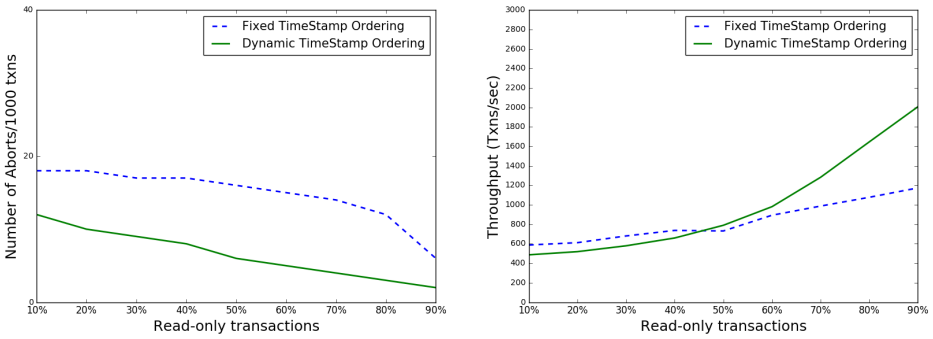
Figure 5.4: Varying concurrency with 99% contention

Varying Concurrency

Next, the performance of the concurrency control protocols is analyzed under varying concurrent number of transactions. The experiment is performed with the high contention ratio of 99:01 and 80% read-only transactions. Figure 5.4 illustrates the results. When the workload is highly concurrent (90 concurrent threads), fixed timestamp ordering has more than 2x aborts compared to the number encountered with dynamic timestamp ordering. This difference is seen because when many transactions are concurrently accessing small set of data, the dynamic timetamping technique adjusts the commit timestamp bounds of concurrent transactions, allowing many of those transactions to commit which would have failed otherwise.

Varying Read-only ratio

In the next experiment, the performance of the concurrency-control protocols is analyzed under varying read-only transaction ratio. With 90% contention and 50 concurrent threads, increase in the read-write transactions (decrease in read-only transactions) leads to an increase in the number of aborts for both protocols. Figure 5.5a illustrates that in the case of increased write transactions, there is high number of aborts for the default



(a) Aborts (b) Throughput

Figure 5.5: Varying read-only ratio with 90% contention

pessimistic fixed timestamp ordering scheme as compared to the dynamic timestamp ordering. Figure 5.5b captures the throughput for decreasing read-only transactions percentage from 90% to 10%. At 10% read-only transactions, fixed timestamp ordering is performing slightly better compared to dynamic timestamp ordering due to the reduced bookkeeping in the former approach. But with higher ratio of read operations, the aborts decrease and throughput increases for both the techniques.

5.6 Related Work

Snapshot Isolation (SI) has been implemented in major database systems, like Oracle and PostgreSQL. Fekete et al. [139] illustrated that SI does not provide serializability and then subsequently Fekete et al. [140] studied the transaction patterns occurring in SI violations. Various techniques have been proposed to make Snapshot Isolation serializable. Some of the techniques [140, 141] perform static analysis of application code and detect SI violation patterns. The potential violations are translated into write-write conflicts, which are then detected by snapshot isolation. However, these techniques are limited in scope and cannot be applied to systems dynamically generating transactions. Cahill et al. [142] develop a SSI (Serializable Snapshot Isolation) methodology to detect

SI violations at run-time and implement the technique over existing snapshot isolation providing database, BerkleyDB. CockroachDB's technique for providing serializable snapshot isolation is inspired by a multi-version timestamp ordering [143] variant proposed by Yabandeh et al. [144].

Various mechanisms have been proposed to reduce aborts in pessimistic as well as optimistic concurrency control (CC) algorithms. For locking algorithms, variants of 2PL such as Order-shared [145] locking, Altruistic Locking [96] and Transaction chopping [146] techniques have been proposed to reduce aborts and achieve higher throughput. Recent works [147] have also explored performing static analysis of application code to order locking requests, such that lock contention is minimized.

Among the lock-free concurrency control schemes, many pessimistic, as well as optimistic techniques employ timestamps to enforce serializable order. The timestamp allocation may be done at the beginning or at the end of the transaction. If a transactional operation violates the timestamp order, the transaction is aborted. Dynamic timestamp allocation [148] schemes have been developed to allocate transactions dynamically, rather than using a fixed timestamp allocation scheme. MaaT [37] employs dynamic timestamps in a distributed setting. MaaT dynamically allocates logical timestamps during validation phase and utilizes soft read and write locks to avoid locking of items during two-phase commit (2PC) between the prepare and commit phase. Our proposed technique builds on MaaT. Tic-toc [135] uses the idea of dynamic timestamp allocation in a single-server setting. BCC [134] (Balanced Concurrency Control) defines essential patterns which occur in non-serializable patterns in Optimistic concurrency control (OCC) algorithms. Rather than aborting a transaction on detecting an anti-dependency, as in OCC, BCC tracks dependencies to abort transactions only when these non-serializable patterns are detected. Although a BCC like technique would reduce the number of aborts, it adds extra overhead to track dependencies.

Deterministic transaction scheduling [86] has also been proposed to eliminate transaction aborts and improve performance under high contention. However, such techniques need a priori knowledge of read-write sets and do not work in an ad-hoc transaction access setting, like in CockroachDB.

CockroachDB's [33] SSI technique employs timestamp allocation, is lock-free and can support distributed transactions. Hence, dynamic timestamping proposed in MaaT is a good fit for CockroachDB, since it is timestamp based and lock-free. The SI implementation in CockroachDB also provides a mechanism to push commit timestamps for distributed transactions. However, this technique cannot be applied to the Serializable variant in CockroachDB, SSI [149] (Serializable Snapshot Isolation).

5.7 Summary

Data access skew is an integral characteristic of modern web applications. A dynamic timestamp ordering based concurrency-control scheme modeled on the MaaT protocol is integrated in the transaction processing mechanism of CockroachDB, to improve its performance under contention. Rather than allocating a fixed commit timestamp at the start of the transaction, the commit timestamp is dynamically allocated, in such a way that it can be best fit on a logical serializable timeline, based on conflicts on the items accessed.

Initial results show that the integration of dynamic timestamp ordering in CockroachDB leads to a decrease in the number of aborts, and a lower drop in throughput in high contention settings. However, the extra processing needed to perform dynamic timestamp allocation, and creation of transaction records for read-only transactions, cause extra overhead.

Integrating dynamic timestamp ordering based concurrency-control in a fully-featured

distributed database like CockroachDB, needed extensive changes at multiple layers. We have implemented and evaluated the changes, and have open-sourced them on Github. To the best of our knowledge, this is the first evaluation of dynamic timestamp allocation technique in a commercial and production cloud database such as CockroachDB.

Part III

Tackling Physical Heterogeneity and Data Processing Variety of IoT Applications

Chapter 6

Data processing architecture for IoT Applications

6.1 Overview

Internet of Things (IoT) applications like smart cars, smart cities and wearables are becoming widespread and are the future of the Internet. Gartner has predicted [150] that there would be 26 billion IoT devices by the end of 2020. These physical devices touch many aspects of life, and have the potential to improve health, commerce and the overall quality of our lives. One of the most challenging aspects for the IoT ecosystem is to manage and get insights from data, which is being continuously collected and transmitted from diverse sensors connected to physical devices [151].

IoT applications continuously collect and process the data received from these devices. One of the major challenges for IoT applications is to efficiently store and process this data. Some of the IoT applications might receive data from a diverse set of sensors. For example, an app might collect both current traffic and weather data to route smart cars. The second fundamental challenge for IoT applications is to integrate the data from a

diverse set of sensors, so that decisions can be made. Third, the data from these diverse set of sensors needs to be continuously monitored for detecting events and anomalies in the physical environment, even in the presence of delayed or missing sensor data.

In this chapter, we focus on designing a data management architecture to efficiently process and store the data for IoT applications. In the next chapter, we will tackle the physical heterogeneity of the sensors and the need to support monitoring on uncertain data from such diverse sensors.

IoT applications and workloads illustrate a wide range of variety in access methodologies. Such applications require to support a high update rate, to support the continuous stream of incoming data from connected sensors, as well as demand real-time analytics on the data being ingested. A heart-rate tracker is an example where continuous ingestion of data, as well immediate access to analytics on that data, is required, so that the application can take actions based on the results of the analytics.

This chapter proposes a data-processing architecture to satisfy the diverse access demands for IoT applications. We design a multi-representation architecture, and tailor it to the need of IoT applications. The data transfer pipeline required to transfer the data between the representations is completely removed, and a deterministic update mechanism is employed to update the representations

6.2 IoT and Multi-Representation Architecture

The multi-representation architecture, described in Section 1.2.1 lends itself well to IoT applications because of the variety of data processing needs of these applications. The multi-representation architecture benefits from the different characteristics of diverse engines. High frequency of incoming updates from connected sensors and simple look-ups of recent values can be handled at OLTP engines like VoltDB, or key-value stores like

Cassandra; continuous event detection can be processed using stream processing engines like Storm; analytics can be performed on engines optimized for analytics, like Vertica and Druid; and batch analytics could be performed by using processing engines such as Spark [152].

However, the need of continuous ETL or CDC pipelines in multi-representation architecture, needed to transfer data, is a big drawback for IoT applications. As IoT applications have to support a very high rate of incoming data from connected sensors, data transfer pipelines consume a high network bandwidth while continuously transferring the data. Additionally, the write-heavy workloads also pose a higher load to support real-time analytics.

Another important characteristic of IoT applications is the need for periodically calculating pre-defined aggregate values. For example, values from a heart-rate or blood sugar-level monitor, might be used to calculate health indexes over a pre-defined past interval and generate alerts if needed. Traditionally data management systems use materialized views to store such pre-defined aggregate values. The views can either be updated asynchronously or synchronously with the update to the base data. If the view is updated asynchronously, the view computation has to be delayed for the update to be applied to the base data to decide on the order of updates. If the view is updated synchronously, the footprint of the update transaction increases, which adversely impacts both system throughput and latency.

To support IoT applications, we need an architecture which can still benefit from the advantages of using different data processing engines, while removing the bottleneck of continuous data transfer, and providing support for specialized IoT analytics like pre-defined aggregates. IoT applications might have some dependencies between incoming data from the sensors, but many updates are independent, and transactional data is not the core characteristic of IoT applications. Because of this reason, and the need to

support pre-defined aggregates and write-heavy workload, we do not employ the data-movement pipeline in Janus, described in Chapter 2, which transfers data at scale, and maintains transactional consistency of data. Based on IoT application characteristics, we employ a solution, where we completely remove the need for data-transfer pipelines.

We propose a *multi-representation* based data processing architecture, where copies of the data are stored in multiple representations. Storing data in different representations will help perform the various update and analytics operations on the representations most suited for them. All representations of the data are updated in a unified manner and hence, completely removing the need to maintain micro-ETL and CDC data pipelines. The different representations can be configured with different schemas to aid in specialized analytical operations like pre-defined aggregates, employed by IoT applications.

A **deterministic ordering** scheme is employed to update the different data representations, rather than updating the representation supporting updates, and then transferring the data between representations. One of the reasons for using micro-ETL and CDC data transfer pipelines is that most systems providing transactional or single-item update support are non-deterministic in nature. Hence, the insert, update and delete requests are first sent to an update processing engine, where the operations are processed in a particular order. The applied updates are then periodically transferred and ingested at other datastores, which support read-only analytical queries. This ensures that the order of updates is the same at all the datastores, and hence analytical queries would return valid results. Non-determinism aids in achieving more concurrency as operations can be processed in any order as long as the transactional or update semantics (serializability, snapshot isolation, atomic updates etc) offered by the given data management system are satisfied. But this leads to the bottleneck of using data transfer pipelines.

A deterministic ordering scheme is suited to IoT applications due to the nature of the data. Many IoT applications either do not need any ordering guarantees, or the ordering

is naturally enforced by the timestamp of the values. However, some IoT applications, might receive data values (corresponding to a physical world state) from multiple sensors, and resolve the final state by a last-writer like approach. In such cases, there is a need for all the representations of the data to have the same final state. The deterministic approach will guarantee that there is one global order enforced at all representations and help in optimizing for latency by removing the data transfer pipelines.

Deterministic ordering has been previously proposed in partitioned [86] and main memory databases [87, 153, 8] to increase performance by reducing aborts using a pre-determined transaction order. We adapt the deterministic scheme to the multi-representation architecture and IoT application characteristics, and use the deterministic order to update the different representations of data. The order of updates will be pre-decided by a sequencer layer and the requests and their corresponding processing order will be then sent to all the data representations. As the update order is pre-defined, each representation can update at its own rate. This allows the decoupling of updates at multiple representations, while removing the need for data transfers between the representations.

To support pre-defined aggregates, different representations may support different schemas. One of the representations can store the entire copy of the data, whereas another representation can be used to store schemas with attributes having aggregate values of base data. Because of the properties of IoT data, and due to the deterministic update mechanism providing a pre-determined update order, the aggregate computation does not have to wait for updates to be applied to base data and can be updated separately from the base data. The decoupling of the aggregate computation from the base data provides the ability to reduce the latency of computation of pre-defined aggregates.

6.3 Architecture

The system architecture is illustrated in Figure 7.2. The data processing architecture comprises of two main components: the Execution Engine (EE) and the Data Storage Layer (DSL). Values from IoT devices are aggregated by sensor aggregators and sent to the application. The IoT application client sends the data management requests (updating the collected data values as well as client read / analytical requests) to the execution engine (EE), which processes the requests and sends them to the data storage layer (DSL). The EE is also responsible for collecting the results of the reads and returning them to the client.

The IoT application client operations are classified into two categories: write transactions and read-only queries. Providing a transaction interface gives the IoT applications the capability to express dependencies, such as values from multiple sensors which should be ingested atomically. Independent events can be expressed as transactions comprising a single operation. A characteristic of IoT applications is that the incoming values from the sensors represent new values and do not depend on existing values as in traditional databases. To model this aspect, each write transaction is comprised of blind writes. EE is responsible for ordering the write transactions. EE sends the deterministic order and the corresponding write transactions to the data storage layer.

A read-only query is specified to be executed at a particular representation. This information is included in the query sent to the EE by the IoT application client. The read-only query is expressed in the query language supported by the representation.

The data storage layer (DSL) stores the data in multiple representations, like row, column, graph and streams. Each representation receives the update transactions as well as read-only queries from the EE. Each representation is responsible for applying the update transactions in the order specified by the EE. As the update order is pre-defined,

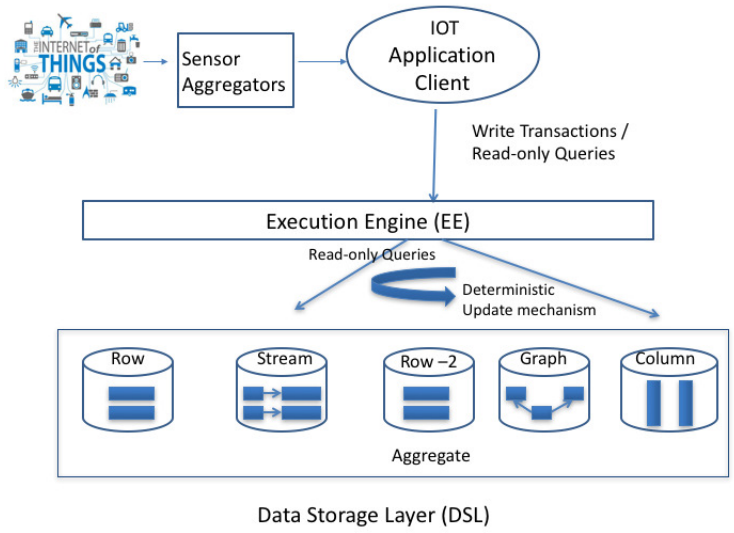


Figure 6.1: System Architecture

each representation can update at their own rate.

Each representation can be integrated with a different existing data storage engines. For example, MySQL can be used as one of the row representation and H-Store can be used as another row representation. Neo4j can be used for graph-based representation.

Each representation can also have a different schema. This characteristic is essential for taking advantage of different representations for efficiently calculating pre-defined aggregates. One representation can store the entire base data, while another representation might store attributes which are aggregates of base data values. As write transactions contain blind writes, pre-defined aggregates and incremental view computation can be computed directly on the aggregated data. The combination of different representations with different schemas and a deterministic update mechanism provides an efficient mechanism to calculate pre-defined aggregates with low latency.

6.4 Deterministic Update Mechanism

Deterministic scheduling is employed to update the data representations. EE receives the transactional write requests from application clients and pre-determines the order of transaction execution. Each representation then applies the transaction writes in the order-specified by the EE. A limitation for the deterministic mechanism is that entire read and write set of a transaction has to be known in advance. However, this aspect does not affect IoT applications, since write transactions are blind writes and the entire transaction (values to write) is known in advance. The deterministic update mechanism comprises 2 stages: sequencing at the EE and applying the ordered updates at each representation.

6.4.1 Deterministic Ordering at the EE

Write transactions are sent to the EE. EE comprises a sequencer component, which determines the order of execution of the write transactions. Each write transaction is assigned a monotonically increasing logical timestamp. The timestamp order assigned by the sequencer is the transaction execution order. The pre-determined execution order is then sent to each representation in the DSL. The sequencer uses a *batching* mechanism to efficiently process the received write transactions, rather than sending each transaction individually to the representations. In particular, the sequencer batches the received transactions, orders them and then sends the batch to each of the representations.

To ensure that the sequencer is not a single point of failure and performance bottleneck, multiple machines are combined to create a highly available sequencer component. As in [86], each machine shares the sequencing in a round-robin fashion, and is responsible for sequencing in the defined epoch period. Using multiple machines for the sequencer helps in the efficiently handling a large volume of write requests.

6.4.2 Applying the pre-determined order

Each representation receives a batch of update transactions from the EE. Each representation is responsible for applying the update transactions in the order specified by the EE. The sequencer sends batches of transactions to each representation. The size of the transaction batch send by the EE is configured for the representations, which are optimized for updates. Some of the data representations are optimized for analytics, and might be overwhelmed by the continuous batches of write transactions sent to them. To overcome this bottleneck, each representation has an additional level of batching to make the writes to the representation more efficient. The batch size at each representation is separately configured, based on the characteristics of the representation.

A challenging aspect of the deterministic scheduling is how to execute the sequence of transactions in the given order efficiently. A naive technique is to use a single thread to process the transaction batch. But this technique limits the throughput achievable by the scheme on modern day multi-core processors. Existing deterministic schemes use different mechanisms to apply the deterministic order in a multi-core setting. H-store [8] partitions the data among the cores and each cores executes the transactions accessing the data under its control, in a single threaded execution. However, the performance is highly dependent on partitioning scheme, as any transaction accessing data across its partition, will lead to stalling the execution to ensure the pre-determined order. Calvin [86] uses a locking mechanism and a locking thread guarantees that access to the data is provided according to the pre-determined transaction order. Bohm [87] is designed for a main memory multi-version environment and divides the concurrency control and execution steps. The concurrency control threads divide the data among them and pre-process the transactions accessing the data under their control and create place holder versions for the transactions to access. The execution threads also divide the data among them

and execute the transactions, blocking when the place holder versions needed have not been created yet. LADS [153] also pre-processes the transactions and examines the transactions for their dependencies, and creates a dependency graph, which captures the dependencies between all the operations (within a transaction and across transactions). It then divides the dependency graph into different sub-graphs while minimizing the edge-cuts among the sub-dependency graphs. The sub-graphs are then executed by different worker threads.

The technique employed in the proposed multi-representation architecture has some similarities to the one employed in LADS, but catered to workloads supported by the IoT applications. The update component of each representation comprises a pre-processing thread and multiple executor threads. To efficiently execute the transactions, the deterministic schedule is applied in a multi-threaded environment in two phases.

Pre-Process

In the first phase, known as the *pre-process* phase, a transaction batch with a pre-determined schedule is pre-processed to construct the dependency graph of transactions within the batch. The dependency graph expresses the ordering dependencies between the transactions present in a given transaction batch. The dependency graph is constructed by the pre-processing thread at the granularity of the transaction batch. Transactions are the nodes in a dependency graph. If any two transactions access a common data item, then they are termed as dependent and the graph has a directed edge from the transaction earlier in the schedule to the one later in the schedule. Each edge is also labelled with the data item(s) causing the dependency. The dependency graph is then uniformly partitioned (each partition gets the same number of vertices) into components known as *batch-partitions*. Each batch-partition is then assigned to a different executor thread. The number of batch-partitions can be configured. Since each partition is assigned to a

different thread, a possible configuration is for the number of partitions to be set to the number of cores in the machine.

Execution

In the second phase, known as the *execution phase*, multiple executor threads are invoked to execute the different batch partitions. Before executing a transaction in the assigned batch partition, the executing thread ensures that all the transactions corresponding to the incoming edges of the transaction have been executed. Each thread executes the batch partition assigned to it. After all the executor threads finish processing, the next transaction batch can be processed.

The proposed technique is catered towards low conflict workloads like write transactions among IoT applications. As opposed to LADS [153], we do not partition the dependency graph for minimum edge-cuts. This would lead to stalls in a high conflict setting, but in a low conflict setting, it reduces the time taken to partition the graph. As we partition the graph dynamically for each batch, the proposed technique would better utilize the CPU, as compared to static partitioning of the data among the threads, employed in H-Store [8].

Representations supporting long running read queries, employ multi-versioning to update the data in parallel. Ingestion of each transaction batch leads to the creation of a new version. The entire transaction batch will be made atomically visible with the creation of a new version. As each transaction comprises blind writes and the transaction batch is made visible atomically, the *execution phase* for applying the deterministic order at such representations is modified to only ingest the last value corresponding to a data item. For executing a transaction in the assigned batch partition, the executor thread does not have to wait for incoming dependencies of a transaction to execute. During the execution of a transaction present in a batch partition, the executor thread first inspects

the outgoing edges of the transaction. If there is an outgoing edge corresponding to write of a data item, then that particular write of the data item is ignored. This ensures that only the last write corresponding to each data item is executed. As the entire transaction batch is made atomically visible, the values after the entire execution of the batch will be consistent.

6.5 Application Scenarios

We now discuss some application scenarios for IoT applications to illustrate the benefits of the system architecture presented.

Weather Monitoring: Consider a weather monitoring application which receives data values corresponding to air pressure, temperature, humidity and wind speed from a diverse set of sensors. The application can be used by clients to execute online queries co-relating different data values, run prediction models and to view the current temperature displayed on the application dashboard.

The proposed system architecture is well suited to such an application. One representation can be a row-based representation backed by a relational engine like MySQL. This representation will store the entire data (with timestamp as the primary key) and will allow clients to write queries co-relating different weather statistics like wind, humidity, temperature etc. The second representation will be used to maintain a pre-defined aggregate like average temperature in a timestamp range (for example a range can be equivalent to 10 readings in base data). The second representation can be updated directly without waiting for the updates to be applied to the first representation and then calculating the average temperature over a range. The average temperature over the most recent range will then be used to update the application dashboard. Since, the second representation needs a simpler data model, we can employ a main memory store

like Redis. A third representation can be used to store the entire data and will be used to execute prediction models. Since, this representation is used to execute offline jobs, it can be backed by a data processing engine like Spark.

Smart City: Consider a future city with self driven cars. These cars are routed based on the weather and traffic conditions, which are monitored using multiple sensors placed between each set of intersections. There are multiple sensor devices on each such road stretch, each sending traffic information and the current weather condition, along with other statistics. The values are sent to the data management system for storage and processing.

The multi-representation architecture can be used for data management of such an application. One of the representation can be a row-based representation (a key-value store like Cassandra can be used) which keeps the current traffic and weather reading for each stretch of the road. This information is used by the application to route traffic. Since, there are application components which query the data and use it to route the traffic, the weather and traffic information is atomically ingested to make sure every decision is made based on a consistent state. A second representation can be used to store the average traffic values in a time-range. Since, the second representation needs a simpler data model, we can use a main-memory store like Redis to store the aggregate. A third representation stores the traffic information over last month and provides the ability to run queries co-relating values at different times (a relational columnar engine can be used). Since, different sensors send information related to the same stretch of the road, a last writer like approach can be used to resolve conflicting values. The deterministic ordering mechanism ensures that all representations write the data in the same order, so every representation will return consistent results. The transactional engine allows the weather and traffic information to be ingested atomically, and using the different representations allows the data management system to handle diverse analytics

and ingestion demands.

6.6 Related Work

Several academic and industrial solutions have been proposed in the past to tackle heterogeneous data processing demands, some of which are akin to current IoT application needs. A detailed discussion of many systems mentioned below is presented in Section 2.7.

There have been several past attempts at using more than one representation for specific needs [21, 43, 57]. Most of the systems target hybrid OLTP-OLAP workloads, and either need a data transfer mechanism or synchronously update the data.

Main-memory database approaches have been proposed to satisfy both OLTP and OLAP demands [56, 54, 59]. These systems also remove the need for data transfer pipelines; by either using a single efficient representation [56, 59] or OS capabilities [54] to separate transaction and analytics. However, these systems concentrate on OLTP and OLAP workloads, and co-locate them. The multi-representation architecture on the other hand, is a solution for diverse operations such as stream processing, graph-based operations, pre-defined aggregates, online queries, batch-processing queries and provide the ability to separate the processing of such diverse operations, while removing the need for transferring data between the systems. SnappyDB [59] supports stream-processing, in addition to enabling execution of OLTP and OLAP workloads in a main-memory setting, but has to choose a single efficient representation for various needs.

The recently proposed Lambda Architecture[78] stores data in two layers, a batching layer (like HDFS) optimized for batch processing and a speed layer, like Apache Storm [154], which processes data streams. However, such an architecture, only deals with simple events and does not support updating multiple values atomically, and does

not provide any ordering mechanism. Furthermore, it deals with stream and batch processing operations, which are only a subset of diverse requirements of IoT applications.

6.7 Summary

IoT applications have diverse data processing needs; from high frequency update support and event detection, to real-time and batch-analytics. We propose a multi-representation based data processing architecture for supporting IoT applications. Data is stored in multiple representations, and a deterministic ordering mechanism is used to update the data, to remove the overhead of data transfer pipelines completely. The proposed data processing architecture allows the ability to perform real-time analytics, tackles diverse requirements of IoT applications, and reduces the latency of computation of pre-defined aggregates.

Chapter 7

M-Stream: A Continuous Monitoring Framework for Uncertain and Diverse Sensor Data

7.1 Overview

With the proliferation of IoT devices, the IoT ecosystem comprises of multiple different physical devices transmitting diverse info about the surroundings. Although data from these different devices might be disparate, it is related to the same physical environment. For example, in a smart home like setting, a surveillance camera and thermostat placed in a room, send different physical attributes related to the same physical location. More value can be extracted from these devices if the data from multiple diverse physical devices can be integrated [155]. One of the major challenges for IoT ecosystem is to process and integrate data from heterogeneous IoT devices, which are collecting and sending disparate data.

Multiple sensors might be transmitting different sensor values related to a physical

event. In such cases, fusing data from multiple sensors gives higher confidence in detecting a physical event. This is also referred to as Sensor fusion [156]. Some such examples can be multiple camera sensors used to detect a person in smart homes like applications, or tracking motion in sports analytics, or integrating data from multiple medical sensors to monitor patients [7]. Additionally, an application should be able to fuse and take decisions based on multiple values from the same sensor. For example, applications collecting vibration data from sensors connected to turbines, want to detect an abnormal vibration only when multiple values are above a threshold [157].

A core challenge faced by IoT applications is handling the underlying uncertainty of the sensor data. Sensor data represents values observed at the devices, and attributes through which we can study the properties of the physical environment related to the sensor. However, the sensed data may have errors, due to the underlying device errors or an error / failure in the communication pipeline. Additionally, sensor data being processed might be delayed or missing due to network connectivity issues, failure of the IoT devices, or due to energy saving mechanisms at IoT devices.

IoT applications also need to continuously monitor and gain real-time insights from such uncertain sensor data. For example, detection of abnormal vibration in turbines or monitoring medical data of a patient needs to be done continuously and in real-time. Over the last decade, data management applications have employed stream processing architectures [158, 159, 19, 18] for continuous real-time processing of data ingested from multiple sources. However, these architectures have some bottlenecks for the IoT data use-case. The push based processing model of stream processing does not fit well with continuous monitoring requirements and the nature of the sensor data. As described above, sensor data can be missing or delayed. Push-based architectures might delay or block processing waiting for delayed or missing sensor data. From the IoT application point of view, processing has to be performed in real-time. Furthermore, these architec-

tures do not provide any abstractions to the application developer to express the integration of sensor data. Any logic for integration or expressing computation over missing and delayed values from the sensors needs to be written by the application developer.

In this chapter, we propose solutions to effectively integrate and continuously monitor uncertain and diverse sensor data. We first propose **Model-based Operators** (MBO), as abstractions to the IoT application developer, to express integration of diverse sensor data. Model-based Operators provide the ability to express the underlying uncertainty of the data, and can integrate data in spatial, temporal or spatio-temporal domains.

We also propose a computation framework named **M-Stream**, to perform continuous monitoring of sensor data. M-Stream provides application developers the ability to use model-based operators to define computations integrating multiple sensor and non-sensor values. In M-Stream, these model-based operators are combined in the form of a dataflow graph, which represents the data processing pipeline of the application. M-Stream continuously executes the model-based operator executions at a defined time interval, to support continuous data processing and monitoring. M-Stream can deal with missing or delayed sensor data, by relying on the model-based operators to express uncertainty in the defined computation, and then utilizing the dataflow graph to ensure that the uncertainty flows through the subsequent computations.

7.2 Designing Model-based Operators

There are many advantages of providing abstractions (model-based operators) to integrate sensor and non-sensor data at the data management layer, rather than writing the logic at the application layer. Offering the model-based operator abstraction at the datastore layer provides the advantage of hiding the complexity of enforcing the guarantee from the IoT application developer. This allows for better modularity and code re-use

as multiple IoT applications can be built using model-based operator abstractions.

Model-based operators are abstractions to define computations, which are defined to be executed successfully, with a certain confidence, based on the model specified by the application developer. We now two present use-cases to show how model-based operators can be in useful.

7.2.1 Use cases

Abnormal vibration detection can be encapsulated as a model-based operator execution. An abnormal vibration in a turbine is detected if a defined percentage of sensor values in a given period are over a pre-defined threshold [157]. The sensor value threshold, the time period and specified percentage for successful execution in this case, can be seen as a model specified by the application developer. The operator is successfully executed if the specified model is satisfied.

Another example of a model-based operation can be a decision to provide drug doses to patients, based on multiple medical sensors calculating, blood pressure, heart rate etc. The application can specify that a drug dose would be given only when the blood pressure is over a particular threshold and heart rate is in a particular range. Such combination of data from multiple sensors can help reduce the uncertainty of the detection and can also hide the underlying errors in values coming from each individual sensor.

7.2.2 Challenges

There are multiple challenges in designing operators for integrating data from sensors and in enforcing the designed model-based operator abstraction guarantee at the datastore layer. We list and discuss some of the challenges below.

- **Exposing the right abstractions for Integration.** One of the biggest challenges is to develop the right abstractions which can help in capturing the use cases of integrating sensor and non-sensor data for IoT applications, and make it easy for the developer to write the business logic.

- **Delayed or Missing Sensor Values.** The sensors continuously send the sensed values to the data management system. But, as discussed above, due to network connectivity issues, or intermittent failures at the sensors, some of the values might be delayed or missing. However, the integration of sensor values needs to be done in real-time, and the application can not wait indefinitely for the sensor values to arrive. The defined abstraction should be able to handle the underlying uncertainty of sensor data.

- **Time Synchronization.** While fusing data from multiple sensors, another challenge is determining which corresponding sensor values to fuse together. If each sensor is synchronized and sending values at the same time, then sensor values with the same timestamp can be fused or integrated together. However if the clocks at sensors are not synchronized, then the integration of the values has to account for the clock offsets. Can there be a model of integrating values from sensors which has clock offsets as a parameter or employs some other form of time synchronization?

7.3 Model-based Operator Abstraction

Model-based operators provide the ability to integrate sensor data in both the spatial and temporal domain. An IoT application developer should be able to express both the integration of values from multiple sensors, as well as values from a single sensor over a time period. In this section we introduce four different Model-based operators to

integrate data over temporal, spatial, and spatio-temporal domains.

Temporal Model-based Operator

We first provide an abstraction to integrate the data from a single sensor in the temporal domain, where a model-based operator $T{-}MBO$ receives a set of values from the sensor s in a time window $[t_s; t_f]$. A model-based operator is said to be executed with confidence c, if at least p percent of the values receiving from the sensor satisfy threshold τ (based on operator σ). $O = \{o_1, ..., o_n\}$ is also a set of outputs of the model-based operator, if the operator is considered to be executed successfully. The outputs can be seen as analogous to writes of a traditional transaction.

Definition: A *Temporal Model-based Operator* is a tuple $T{-}MBO$ =$(s, V, t_s, t_f,$ $\mathcal{T}, \gamma, \tau, \sigma, p, c, \varphi, O)$ where

1. s is a single sensor,
2. $V \subset \mathbb{R}$ is a set of (sensor) values,
3. t_s and t_f $(t_s \leq t_f)$ are two real valued variables (time-points) representing the boundary of time window,
4. $\mathcal{T}$ is a finite set of time-points where for each $t \in \mathcal{T}, t_s \leq t \leq t_f$,
5. $\gamma : \mathcal{T} \rightarrow V$ is a partial mapping that assigns value to time-points,
6. $\tau \in \mathbb{R}$ is a threshold,
7. $\sigma \in \{<, >, \leq, \geq, =, \neq\}$ is an inequality operator,
8. $p, c \in [0, 1] \subset \mathbb{Q}$ are the percentage and confidence,
9. φ is a (user-defined) mapping that returns confidence c based on the time-points $\mathcal{T}$, mapping γ, threshold τ, operator σ and percentage p, and

10. O is a set of outputs.

For example, in case of vibration detection use-case described earlier, the application developer wants the detection to be made based on values received from a single sensor over a time period. Consider a time window $[1, 9]$, and set of time-points $\mathcal{T} = \{2, 4, 6, 8\}$ where $\gamma(2) = 5$, $\gamma(4) = 7$, $\gamma(6) = 10$, and $\gamma(8) = 6$ are the received values. Let threshold $\tau = 8$ and operator "$<$" says that a sensor value satisfies the threshold if it is less than the threshold. Let $p = 0.7$ and function φ returns $\frac{|V|_{<\tau}}{|\mathcal{T}|}$ (ratio of values that satisfy the threshold τ to the number of time-points) as the confidence. Here, since in 0.75 of the time-points (time-points 2, 4, and 8) the threshold condition is satisfied, the model-based operator executes successfully and φ also return 0.75 as the confidence. The definition of φ given here, is one way of computing the confidence. Applications might use a different definition based on the context.

Spatial Model-based Operator

The next model based operator is defined to integrate data over the spatial domain. For integrating over the spatial domain, we want the ability to integrate the values from multiple sensors. Values from different sensors can represent values from different related locations, like soil moisture readings from different locations in a farm. Alternatively, they might provide information about different physical attributes of a single location. An example of this case is multiple sensors in a room, sending temperature, humidity and air-pressure respectively. Another example of integrating data from multiple sensors in the scenario with medical dataset explained above. Data from sensors sending information patient's blood pressure, heart rate, insulin level etc, might be integrated to perform continuous health monitoring and administer medicines.

The system is composed of n sensors, s_1, s_2, ..., s_n. Each sensor s_i has a value associated with it, v_i, which is emitted periodically. A spatial model-based operator

$S{-}MBO$ is said to be executed with confidence c, if at least p percent of the n sensors values satisfy their defined thresholds. Note that spatial model-based operators can fuse non-sensor inputs with sensor inputs as well. A non-sensor input such as a data-item with key k and value val can be represented as input from sensor k with value val.

Definition: A *Spatial Model-based Operator* is a tuple $S{-}MBO = (S, V, \gamma, T, \tau, \sigma, p, c, \varphi, O)$ where

1. S is a set of sensors,
2. $V \subset \mathbb{R}$ is a set of (sensors) values,
3. $\gamma : S \rightarrow V$ is a partial mapping that assigns values to sensors,
4. $T \subset \mathbb{R}$ is a set of thresholds,
5. $\tau : S \rightarrow T$ is a total mapping that assigns thresholds to sensors,
6. $\sigma : S \rightarrow \{<, >, \leq, \geq, =, \neq\}$ is a total mapping that assigns operators to sensors,
7. $p, c \in [0, 1] \subset \mathbb{Q}$ are the percentage and confidence,
8. φ is a (user-defined) mapping that returns confidence c based on sensors S, mappings γ, τ, and σ, and percentage p, and
9. O is a set of outputs.

Spatio-Temporal Model-based Operators

The last two model based operators are defined to integrate data over both spatial and temporal domain where the system is composed of n sensors, s_1, s_2, ..., s_n and each sensor s_i has a set of values $v_{i1}, v_{i2}, ..., v_{im}$ over m time-points. We define two different operators $ST{-}MBO$ and $TS{-}MBO$ where in $ST{-}MBO$ we first integrate data over

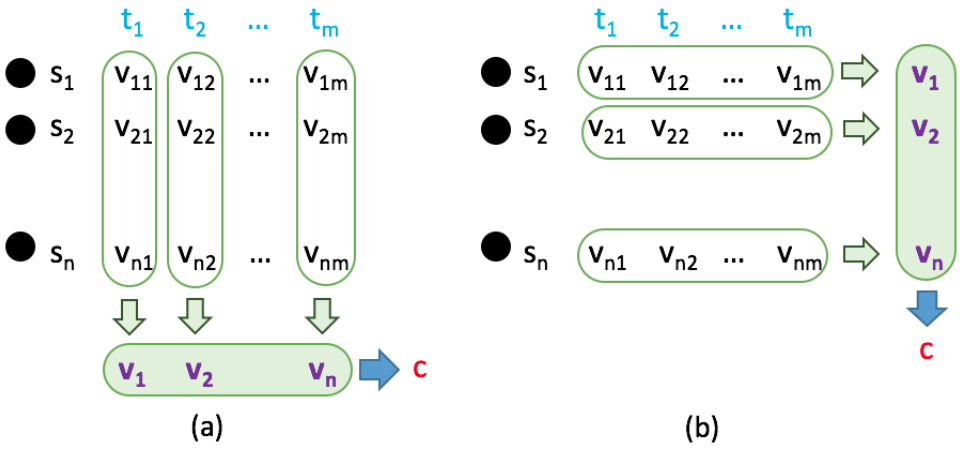


Figure 7.1: Spatio-temporal Operators

spatial domain and then over temporal domain, and in $TS{-}MBO$ we first integrate data over temporal domain and then over spatial domain.

In $ST{-}MBO$, as can be seen in Figure 7.1(a), a user-defined function ψ integrates data in each time-point t_i and returns some value v_i. Then similar to the temporal model-based operator, the operator is said to be executed with confidence c, if at least p percent of the values returned by the function ψ satisfy threshold τ (based on operator σ).

Definition: A *Spatio-Temporal Model-based Operator* is a tuple $ST{-}MBO = (S, V, t_s, t_f, \mathcal{T}, \gamma, \tau, \sigma, P, c, \psi, \varphi, O)$ where

1. S is a set of sensors,
2. $V \subset \mathbb{R}$ is a set of (sensor) values,
3. t_s and t_f are two real valued variables (time-points) representing the start and the end of the time window,
4. $\mathcal{T}$ is a finite set of time-points where for each $t \in \mathcal{T}, t_s \leq t \leq t_f$,

5. $\gamma : S \times \mathcal{T} \rightarrow V$ is a partial mapping that assigns a value to each sensor at each time-point,

6. $\tau \in \mathbb{R}$ is a threshold,

7. $\sigma \in \{<, >, \leq, \geq, =, \neq\}$ is an inequality operator,

8. $p, c \in [0, 1] \subset \mathbb{Q}$ are the percentage and confidence,

9. ψ is a (user-defined) mapping that aggregates values in each time point and returns a single value,

10. φ is a (user-defined) mapping that returns confidence c based on the mapping ψ, threshold τ, operator σ and percentage p, and

11. O is a set of outputs.

In $TS{-}MBO$, as can be seen in Figure 7.1(b), we assign a separate threshold and percentage to each sensor and similar to temporal operator compute a value corresponding to each sensor using the user-defined function ψ. These confidences then aggregate over the spatial domain using another user-defined function φ that returns a value for confidence c.

Definition: A *Spatio-Temporal Model-based Operator* is a tuple $TS{-}MBO = (S, V, t_s, t_f, \mathcal{T}, \gamma, T, \tau, \sigma, P, c, \varphi, \psi, O)$ where

1. S is a set of sensors,

2. $V \subset \mathbb{R}$ is a set of (sensor) values,

3. t_s and t_f are two real valued variables (time-points) representing the start and the end of the time window,

4. $\mathcal{T}$ is a finite set of time-points where for each $t \in \mathcal{T}, t_s \leq t \leq t_f$,
5. $\gamma : S \times \mathcal{T} \rightarrow V$ is a partial mapping that assigns a value to each sensor at each time-point,
6. $T \subset \mathbb{R}$ is a set of thresholds,
7. $\tau : S \rightarrow T$ is a total mapping that assigns thresholds to sensors,
8. $\sigma : S \rightarrow \{<, >, \leq, \geq, =, \neq\}$ is a total mapping that assigns operators to sensors,
9. $P : S \rightarrow [0, 1] \subset \mathbb{Q}$ is a total mapping that assigns percentage to sensors,
10. $c \in [0, 1] \subset \mathbb{Q}$ is the confidence,
11. ψ is a (user-defined) mapping that returns a value (confidence) for each sensor based on mappings γ, τ, and σ, and percentages P,
12. φ is a (user-defined) mapping that aggregates values from different sensors (resulted from ψ) and returns a confidence value c, and
13. O is a set of outputs.

7.4 M-Stream Design

M-Stream is a computation framework, which aims to provide the IoT application developer, the ability to efficiently and easily write the business logic of the application, define computations on continuous streams of data, and be able to handle the underlying uncertainty of the data. First, we given an overview of how the M-Stream framework combines the model-based operators to design a computation pipeline for continuous

data processing and monitoring of IoT applications. An instance of M-Stream is then described to illustrate its working.

7.4.1 M-Stream Overview

M-Stream is a computation framework, designed to support the continuous real-time data processing and monitoring needs of IoT applications. M-Stream employs the model-based operation abstraction to integrate sensor and non-sensor data, and provides the ability to combine the defined model-based operation executions in a data processing pipeline, to represent the computational needs of IoT applications. To support the requirement for continuous data processing, M-Stream employs a continuous processing model to compute the model-based operations periodically after a defined time period. M-Stream combines the continuous processing model with the traditional push-based stream processing model to perform computations. M-Stream is designed to be fault-tolerant, to be able to deal with missing or uncertain sensor data. It utilizes the confidence in the computation of model-based operations to capture the uncertainty of the data and the decisions made. M-Stream also provides the ability to integrate a prediction model within the data processing pipeline.

We now illustrate how M-Stream operates. First, the computation model of M-Stream is discussed briefly. Then, we abstractly describe the integration of the model-based operations within M-Stream's computational model. We then discuss how M-Stream can be integrated with a statistical prediction model to handle missing or delayed sensor values.

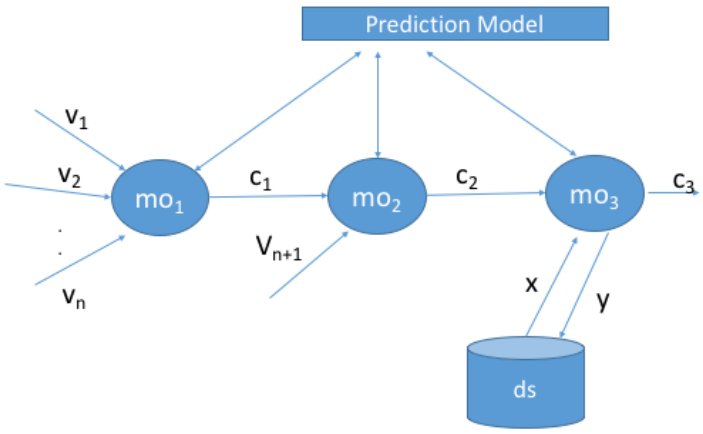


Figure 7.2: M-Stream's computational model: Dataflow architecture employing model-based operator abstraction

Computational Model

The data management pipeline in M-Stream consists of incoming values from multiple sensors, and persistent sources like datastores. The pipeline can be represented as a directed acyclic graph of computation. Each node in the graph represents a computation. For now, lets assume that each computation is represented by a model-based operator. Each node has incoming and outgoing edges. The incoming edges are inputs to the model-based operator, which can be inputs from the sensors or inputs from other model-based operators. The outgoing edges are the outcome of the model-based operations, including, whether the model-based operation is successful and the confidence in its success.

Figure 7.2 illustrates a dataflow graph architected using model-based operators. mo_1, mo_2 and mo_3 represent model-based operators. $v_1..v_{n+1}$ represent inputs from the sensors s_1, .. s_{n+1} respectively. ds is a datastore which stores some key and value data items. x and y are two such items. x is an input to mo_3, and y is an output of mo_3. c_1, c_2 and c_3 represent the confidence of the model-based operator executions.

Each model-based operator can be performing either a spatial, temporal or a combined spatio-temporal integration of sensor values. Each model-based operator is com-

puted continuously after a defined time period. The periodic computation is performed to support the continuous data processing and monitoring needs of IoT applications. Once all the inputs to the model-based operator arrive, and after the pre-defined period, the model-based operator is executed. Once the operator is computed, the confidence in the operator execution is produced as output. This can be an input to other model-based operator executions. Additionally, the operator execution can have other outputs, like writing to a persistent datastore. The reading and writing from the persistence store can be executed as a transaction, to ensure that other concurrent operations accessing the datastore have not modified the accessed items. Some of the applications might require transaction isolation guarantees for their reliability. For ensuring that corresponding values are fused together, M-Stream checks the timestamps corresponding to sensor values, and ensures that the values fused together are not more than a defined delta time period apart. M-Stream does not deal with time synchronization, and assumes an outside entity is responsible for this purpose. If the timestamp corresponding to the sensor value being fused, is more than delta apart from the timestamp of the data fusion, the value is considered to be missing, and is treated as defined in the section below.

In Figure 7.2, mo_1 integrates values from sensors s_1, .. s_n and outputs the confidence of the outcome, c_1 as output. mo_2 integrates the value from s_{n+1} and outcome of mo_1, and outputs the confidence of the outcome of the execution, c_2. mo_3 integrates a non sensor output, with the confidence of the outcome of mo_2. If mo_3 succeeds, which is determined by the thresholds specified in the abstraction defining mo_3, then data item y will be updated in the datastore ds.

Delayed or Missing Sensor data

As described earlier, one of the major challenges for sensor data integration is that some of the values needed for integration of sensor values can be delayed or missing.

As M-Stream employs model-based operators to encode the computations needed by the applications, it is well suited to handle delayed or missing sensor data. The confidence of the model-based operator execution can be used to reflect the uncertainty of the underlying data and decisions. For example, suppose the t-MBO operator temporaly integrates data from sensors, $s..1$, $..s..n$. Consider the confidence being computed as a function returning the ratio of sensor values that satisfy the threshold to the number of values fused together. If at the time of a periodic t-MBO computation, the value from sensor s_i is missing, or reflects a stale value (based on the delta time period specification by the application developer), then the model-based operator can continue processing assuming that s_i does not satisfy the threshold specified for the successful operator execution. Even if the execution is successful, the reduced confidence of the operator execution will reflect the increased uncertainty in the occurrence of the physical event represented by the defined t-MBO.

Additionally, statistical prediction models can also be used to help in dealing with missing or delayed values. A prediction model can be used to predict any missing input to the node at a particular timestamp, with a probability. If some of the values are missing or delayed, and do not arrive up-to a defined grace period after the periodic computation interval, the operator is executed using the predicted values. The estimation of the predicted value being above or below the threshold (returned by the statistical model) defined in the model-based operator definition, is then integrated in the confidence of the execution. If the prediction model reflects that there is a low probability of a missing input to satisfy the threshold condition in the model-based operator execution, then the confidence of the operation execution will reflect that.

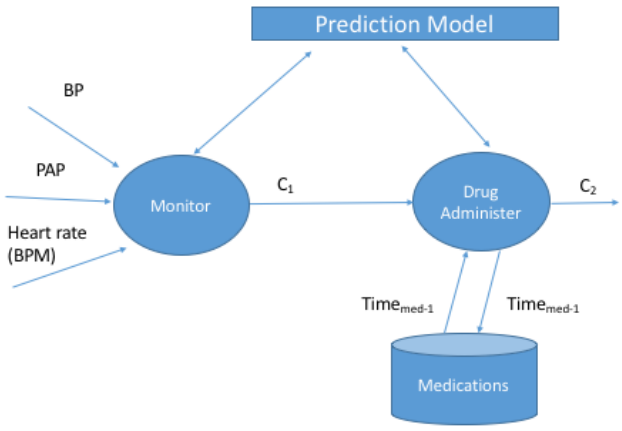


Figure 7.3: An instance of M-Stream's dataflow for Patient Monitoring

7.4.2 An Instance of M-Stream

Figure 7.3 shows an instantiation of M-Stream for patient monitoring in a futuristic setting. The setting is similar to the one envisioned by other works handling healthcare data [7, 160]. Sensors connected to the patient continuously send information such as blood pressure, PAP, heart rate and other metrics, which has to be continuously monitored, and is used for administering drugs and raising alerts and notifications. The application developer can use a model-based operator to integrate the data from these sensors, and based on thresholds decide the confidence in administering the drug (monitor operator in the figure). The drug administer operator takes the confidence of the prediction, and the last time the medication (and any medication which should not be administered together with this medication) was given an input, and decides whether the drug should be administered. The medication datastore is then updated. Note that there might be other dataflows monitoring other parameters, and writing to the medication datastore. Hence, the drug administer operation should be executed with transaction isolation guarantees. Secondly, we have to ensure that even on failures, "exactly-once" guarantee of execution is provided. Since, the sensors might fail or have delays in sending data, each computation can access a prediction model, which can be used to predict the

missing or delayed values, and execute the defined operations periodically in spite of such errors. Both the monitor and drug administer operators also output the confidence of their execution. This confidence is used as input to other operators, which ensures that underlying uncertainty in the sensor data flows through the graph.

7.5 Related Work

Traditional database systems expose transactions [161] as abstractions to the application developer. The application developer can express that an event would be considered to be executed only if all the sub-events comprising the transaction take place atomically. Stored procedures [162] are used by developers to express a set of functions to be executed on the data. They allow the application developer to express the business requirements, and consolidate logic and computations at the data management layer. Triggers [163] in databases allow a set of functions to be executed, in response to a certain event, like data insertion. Model-based Operators, like stored procedures and triggers, are executed given that a set of defined conditions are met, and allow for code-reuse, ease in programability, and pushing the computation to the data management layer. However, each model-based operator execution also outputs a confidence in the execution, to reflect the uncertainty of the underlying data. Model-based Operator abstraction is designed to deal with uncertain data, to cater to IoT applications, and does not need any specifications to be provided if the data is missing or delayed. Confidence of computation can used to reflect the uncertainty of the data, which can be utilized by combining the model-based operators, as done in M-Stream.

Stream processing architectures [158, 18, 19] have been employed to perform continuous real-time data processing. Stream processing architectures like Storm [19] use a dataflow architecture and push-based processing, to perform computation and push the

results, once the data arrives. M-Stream builds on the stream-processing model and defines each node in the dataflow graph to be executed at defined time periods, so as to not block for missing or delayed sensor values. And each computation node in the dataflow graph is expressed using a model-based operator. M-Stream can be implemented over the top of existing open-source distributed stream-processing architecture like Storm, as to reuse the basic dataflow constructs and communication between components. Architectures to combine traditional transaction processing and stream processing have been proposed as well [164]. Like the traditional stream processing architectures, they are not suitable for handling the underlying uncertainity of sensor data.

Previous work in the context of wireless sensor networks, have proposed techniques for probabilistic event detection [165], efficient data collection [166, 167] and approximate querying over sensor data [168], to reduce communication with sensor devices for energy efficiency. M-Stream is build for an environment where sensor data is being continuously transmitted and collected, but can be delayed or missing. Some of the proposed probabilistic models [168, 169] could be used to design Statistical models in M-stream, to provide probabilistic values of missing data, which can be utilized in model-based operator executions.

7.6 Summary

IoT applications need to continuously monitor incoming data from a diverse set of sensors, and handle the uncertainty of sensor data. We propose the initial design of M-Stream, which is a novel computation framework, to enable continuous monitoring and real-time data-processing of sensor data. M-Stream can enable the developers to employ a continuous monitoring architecture, by combining model-based Operators, which are abstractions to the application developer to express computation over sensor data.

Model-based operators can handle the uncertainty of the sensor data, by allowing the developer to express the computation to be successful, based on a subset of conditions expressed over spatial and temporal domains. The developer can also define a function to compute the confidence in each model-based operator execution, which can reflect the errors or uncertainty in the data fused in the operator execution. M-Stream combines these operators in a dataflow graph, with each node of the graph expressing the computation as a model-based Operator.

Part IV

Concluding Remarks

Chapter 8

Conclusion and Future Work

Data variety is a core challenge posed by modern-day data-intensive applications. In this dissertation, we have proposed solutions for tackling diverse challenges posed by variety of the data, workloads processing such data, and heterogeneity in the data infrastructure supporting the processing. Specifically, we tackle problems related to *variety of structure and access workloads*, *distribution heterogeneity within workloads*, and *physical heterogeneity and data processing variety of IoT* applications.

Variety in structure of the data and diverse access methodologies is one of the essential elements of the data management ecosystem. This variety in structure and the different accesses, has resulted in heterogeneity at the data infrastructure level. Applications now employ a multitude of data management systems with different underlying representations for their data processing needs. The resultant multi-representation architecture can individually cater to many workloads, but has many challenges. This dissertation tackles some of these challenges and proposes solutions: 1) to provide support for consistent real-time analytics while supporting a high throughput of transactions, and 2) defining consistency semantics in a multi-representation architecture setting.

In this dissertation, we propose Janus, a hybrid multi-representation cloud datastore,

to efficiently support both transactions and diverse, consistent real-time analytics. Janus uses multiple representations to support a high throughput of transactions, as well as support diverse real-time analytics. Transactions and analytics are executed on different representations, but the novel data-movement pipeline in Janus guarantees that each analytical query observes the same order of transaction execution, and does not observe the partial impact of any transaction. Janus is a cloud datastore and supports large-scale data via partitioning, and allows distributed transactions as well as different partitioning strategies, to support diverse characteristics of representations. Experiments on AWS, a public cloud platform, demonstrate that Janus can support consistent real-time analytics at scale.

For better understanding of consistency semantics over heterogeneous data representations, we propose Typhon, which defines a formal consistency model for data split across representations. The consistency model in Typhon uses a notion of entities for relating data items and defines dependencies at the granularity of an entity. These dependencies capture the happens-before relationships for accesses to data across representations. The model is integrated with the traditional conflict-graph model, and can be used to analyze the consistency guarantees provided by existing protocols. We also design a protocol, Cerberus, which enforces the implicit-casual dependencies introduced in Typhon. Cerberus operates in a single-entity transaction model, which is ideal for social media and other such applications, which require consistency guarantees at the granularity of an entity, but do not want to employ a general purpose transaction model across their entire dataset. We perform extensive experimental evaluation to demonstrate that Cerberus is scalable, and offers a practical solution for providing consistency in a multi-representation setting. It can prevent consistency violations by only adding a small performance overhead, as compared to a solution providing no consistency guarantee across multiple representations.

To deal with distribution heterogeneity within the workloads, this dissertation studies the impact of workloads with skew in access distribution (high contention), and operation-type distribution (read-heavy). The impact of such workloads is studied in the context of a distributed database setting in CockroachDB, an open-source distributed database, which supports large-scale data processing via data distribution and replicas, which are synchronized using a consensus layer. We propose protocols to scale read-heavy workloads in consensus protocols like Raft. Evaluation results demonstrate that the proposed strongly-consistent quorum reads, and their combination with existing leader-replica based reads, leads to an improvement in both throughput and latency, and provides a configurable trade-off between read and write latencies. For supporting high contention in a distributed OLTP database, we integrate an existing dynamic timestamp allocation scheme in CockroachDB. To the best of our knowledge, this is the first study of dynamic timestamp allocation like technique in an open-source commercial database like CockroachDB. Initial results bring out the performance improvements of using dynamic timestamping under high contention, and the overheads which need to be tackled to deploy such solutions in production settings.

This dissertation also analyzes the issue of variety in the context of IoT applications, which have to handle both the physical heterogeneity of diverse sensors, which capture the different elements of the physical environment, as well as different data processing needs. We propose a multi-representation architecture catered to IoT applications, which removes the data transfer pipeline, in order to better support IoT workloads. Using deterministic scheduling to update all the representations ensures that data transfer pipelines are not needed. The resultant architecture can support computation of predefined aggregates with low latency, by separating their update processing from the update to base data, and by using different schema for the representation supporting the aggregate storage and computation.

We finally present the initial design of M-Stream, a computation framework for monitoring uncertain and diverse sensor data. Model-based operators are proposed as abstractions to the IoT application developer, to spatially and temporally integrate sensor data. A model-based operator execution also expresses the underlying uncertainty of integrated data. The proposed design of M-Stream combines the model-based operators in the dataflow graph, and employs a continuous processing model to cater to monitoring requirements of IoT applications. We illustrate the utility of the architecture, by describing some application scenarios, and demonstrating how M-Stream, and the underlying model-based operators, can be used to architect a real-time monitoring pipeline.

8.1 Future Work

This dissertation presents solutions to critically understand and tackle several different problems related to data variety. There are still many problems that need to be addressed, to efficiently cater to an increasing set of variety challenges posed by modern-day applications.

Query execution in multi-representation architectures poses several novel challenges. Currently, even when multiple representations are employed, the application developer has to choose between the different representations, either at the granularity of the access workload or for each query. However, this decision might not be always trivial. Apart from the consistency trade-offs between primary and secondary representations, the most efficient representation in terms of execution time might not be easy to choose. Furthermore, for larger queries, executing part of the query at different representation might be beneficial. A possible line of research is to analyze and use traditional query optimization [170] like techniques for choosing the best representation for an incoming query, or break the query into different fragments, to be executed at different representations.

We propose Typhon, which defines implicit causal guarantees for transactions accessing data across representations. The consistency model in Typhon is valid for general-purpose transactions, where each transaction accesses data related to multiple entities. Cerberus presents a protocol in the single-entity transaction model, allowing it to offer a practical solution for applications which require guarantees only at the granularity of an entity. However, some applications might need to support transactions accessing more than one entity across representations. A future line of research is to evaluate and study applications with such requirements, and attempt to build protocols to provide Typhon's consistency guarantees in a general transaction model. Analysis needs to be done to evaluate whether the resultant protocol(s) can provide such guarantees at a lower cost, as compared to approaches providing global serializability across all the data representations.

This dissertation proposes and studies separate solutions for tackling read-heavy and high-contention workloads respectively. The strongly-consistent quorum reads operate at the consensus layer, and enables reading the latest value without going to the leader. For supporting high contention transaction workloads, we integrate dynamic timestamp allocation at the concurrency-control layer. The two proposed solutions can be integrated to offer a solution which can address both aspects of distribution heterogeneity together. Both proposed solutions are compatible as well. Transactions in CockroachDB currently perform read at the lease-holder replica. However, after the integration of the two solutions, multiple reads of the transaction can be performed from non-lease holder nodes. Dynamic timestamp ordering technique allocates the commit timestamp at the end of the transaction, and can verify whether versions read at non-leader nodes are still valid.

In the line of research for handling the physical heterogeneity of IoT applications, the next step is to build and implement M-Stream. M-Stream can be implemented on top of a stream-processing engine like Storm. Model-based operators can be built using

bolts (execution nodes), and incoming sensor data could be ingested into the system using Kafka [44] queues and spouts (data generation nodes). Additionally, the underlying model-based operator abstraction can be explored further. The current design of the abstraction is well suited to event detection, which is a common scenario for IoT applications. Further work needs to investigate whether more generic operators can be defined, which can model statistical aggregation like use-cases. For such cases, the confidence function could be expressed by providing confidence intervals for aggregate output values, rather than the current form of model-based operator execution, where the confidence is expressed as a single value between 0 and 1.

Bibliography

[1] S. Sagiroglu and D. Sinanc, *Big data: A review*, in *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, pp. 42–47, IEEE, 2013.

[2] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, *et. al.*, *Tao: Facebook's distributed data store for the social graph.*, in *USENIX ATC Annual Technical Conference*, pp. 49–60, 2013.

[3] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, *E-store: Fine-grained elastic partitioning for distributed transaction processing systems*, *Proceedings of the VLDB Endowment* **8** (2014), no. 3 245–256.

[4] J. R. Ferrari, T. R. Lookingbill, and M. C. Neel, *Two measures of landscape-graph connectivity: assessment across gradients in area and configuration*, *Landscape ecology* **22** (2007), no. 9 1315–1323.

[5] "Farmville Case-Study.." `http://www.adweek.com/digital/farmville-about-to-cruise-past-80-million-users/`, 2010.

[6] "Michael Jackson death almost takes Internet with him." `http://www.cnn.com/2009/TECH/06/26/michael.jackson.internet`, 2009.

[7] N. Bui and M. Zorzi, *Health care applications: a solution based on the internet of things*, in *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*, p. 131, ACM, 2011.

[8] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, *The end of an architectural era:(it's time for a complete rewrite)*, in *Proceedings of the 33rd international conference on Very large data bases*, pp. 1150–1160, VLDB Endowment, 2007.

[9] "VoltDB." `http://voltdb.com/`, 2018.

[10] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, *et. al.*, *H-store: a high-performance,*

distributed main memory transaction processing system, *Proceedings of the VLDB Endowment* **1** (2008), no. 2 1496–1499.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, *Bigtable: A distributed storage system for structured data*, *ACM Transactions on Computer Systems (TOCS)* **26** (2008), no. 2 4.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, *Dynamo: amazon's highly available key-value store*, in *ACM SIGOPS operating systems review*, vol. 41, pp. 205–220, ACM, 2007.

[13] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, *The vertica analytic database: C-store 7 years later*, *Proceedings of the VLDB Endowment* **5** (2012), no. 12 1790–1801.

[14] "MonetDB." `http://monetdb.com/`, 2018.

[15] S. I. F. G. N. Nes and S. M. S. M. M. Kersten, *Monetdb: Two decades of research in column-oriented database architectures*, *Data Engineering* **40** (2012).

[16] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, *et. al.*, *C-store: a column-oriented dbms*, in *Proceedings of the 31st international conference on Very large data bases*, pp. 553–564, VLDB Endowment, 2005.

[17] "Neo4j." `http://neo4j.com/`, 2018.

[18] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et. al.*, *Storm@ twitter*, in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 147–156, ACM, 2014.

[19] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, *Twitter heron: Stream processing at scale*, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 239–250, ACM, 2015.

[20] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, *et. al.*, *System r: relational approach to database management*, *ACM Transactions on Database Systems (TODS)* **1** (1976), no. 2 97–137.

[21] "Oracle 12c." `http://www.oracle.com/us/corporate/features/database-12c/index.html`, 2018.

[22] "Microsoft SQL Server 2017." https://www.microsoft.com/en-us/sql-server/sql-server-2017, 2018.

[23] "MySQL." https://www.mysql.com/, 2018.

[24] D. J. Abadi, S. R. Madden, and N. Hachem, *Column-stores vs. row-stores: how different are they really?*, in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 967–980, ACM, 2008.

[25] G. P. Copeland and S. N. Khoshafian, *A decomposition storage model*, in *ACM SIGMOD Record*, vol. 14, pp. 268–279, 1985.

[26] "Vertica." http://vertica.com/, 2018.

[27] S. Das, C. Botev, K. Surlaker, B. Ghosh, B. Varadarajan, *et. al.*, *All aboard the databus!: Linkedin's scalable consistent change data capture platform*, in *Proc. of SoCC*, p. 18, 2012.

[28] D. Ongaro and J. Ousterhout, *In search of an understandable consensus algorithm*, in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 305–319, 2014.

[29] L. Lamport, *The part-time parliament*, *ACM Transactions on Computer Systems (TOCS)* **16** (1998), no. 2 133–169.

[30] L. Lamport *et. al.*, *Paxos made simple*, *ACM Sigact News* **32** (2001), no. 4 18–25.

[31] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, *et. al.*, *Spanner: GoogleâĂŹs globally distributed database*, *ACM Transactions on Computer Systems (TOCS)* **31** (2013), no. 3 8.

[32] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, *et. al.*, *F1: A distributed sql database that scales*, *Proceedings of the VLDB Endowment* **6** (2013), no. 11 1068–1079.

[33] "CockroachDB." https://www.cockroachlabs.com/, 2018.

[34] V. Arora, F. Nawab, D. Agrawal, and A. E. Abbadi, *Janus: A hybrid scalable multi-representation cloud datastore*, *IEEE Transactions on Knowledge and Data Engineering* **30** (April, 2018) 689–702.

[35] V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi, *Typhon: Consistency semantics for multi-representation data processing*, in *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pp. 648–655, IEEE, 2017.

[36] V. Arora, T. Mittal, D. Agrawal, A. Abbadi El, X. X, Zhiyanan, and Zhujianfeng, *Leader or majority: Why have one when you can have both? improving read scalability in raft-like consensus protocols*, in *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, USENIX Association, 2017.

[37] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi, *Maat: Effective and scalable coordination of distributed transactions in the cloud*, *Proceedings of VLDB* **7** (2014), no. 5 329–340.

[38] V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi, *Multi-representation based data processing architecture for iot applications*, in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pp. 2234–2239, IEEE, 2017.

[39] "Gartner. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation." `https://www.gartner.com/doc/2657815/`, 2014.

[40] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, *Megastore: Providing scalable, highly available storage for interactive services.*, in *CIDR*, vol. 11, pp. 223–234, 2011.

[41] H. Jafarpour, J. Tatemura, and H. Hacigümüs, *Transactional replication in hybrid data store architectures.*, in *EDBT*, pp. 569–580, 2015.

[42] P.-Å. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, *et. al.*, *Real-time analytical processing with sql server*, *Proceedings of the VLDB Endowment* **8** (2015), no. 12 1740–1751.

[43] R. Ramamurthy, D. J. DeWitt, and Q. Su, *A case for fractured mirrors*, *The VLDB Journal* **12** (2003), no. 2 89–101.

[44] J. Kreps, N. Narkhede, J. Rao, *et. al.*, *Kafka: A distributed messaging system for log processing*, in *Proceedings of the NetDB*, pp. 1–7, 2011.

[45] P. A. Bernstein and N. Goodman, *Concurrency control in distributed database systems*, *ACM Computing Surveys (CSUR)* **13** (1981), no. 2 185–221.

[46] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Data structures for disjoint sets*, *Introduction to Algorithms* (2001) 498–524.

[47] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, *Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration*, *Proceedings of the VLDB Endowment* **4** (2011), no. 8 494–505.

[48] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, *Benchmarking Cloud Serving Systems with YCSB*, in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010.

[49] "Amazon AWS." `https://aws.amazon.com/`, 2018.

[50] "Protocol Buffers." `https://developers.google.com/protocol-buffers/`, 2018.

[51] D. L. Mills, *Network time protocol (ntp)*, *Network* (1985).

[52] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, *Hyrise: a main memory hybrid storage engine*, *Proceedings of the VLDB Endowment* **4** (2010), no. 2 105–116.

[53] J. Arulraj, A. Pavlo, and P. Menon, *Bridging the archipelago between row-stores and column-stores for hybrid workloads*, in *Proceedings of the 2016 International Conference on Management of Data*, pp. 57–63, 2016.

[54] A. Kemper and T. Neumann, *Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots*, in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pp. 195–206, 2011.

[55] V. Sikka, F. Färber, W. Lehner, S. K. Cha, *et. al.*, *Efficient transaction processing in sap hana database: the end of a column store myth*, in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 731–742, 2012.

[56] H. Plattner, *A common database approach for oltp and olap using an in-memory column database*, in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 1–2, 2009.

[57] I. Alagiannis, S. Idreos, and A. Ailamaki, *H2o: a hands-free adaptive store*, in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 1103–1114, ACM, 2014.

[58] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu, *Es 2: A cloud data storage system for supporting both oltp and olap*, in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pp. 291–302, IEEE, 2011.

[59] J. Ramnarayan, S. Menon, S. Wale, and H. Bhanawat, *Snappydata: a hybrid system for transactions, analytics, and streaming*, in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pp. 372–373, ACM, 2016.

[60] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav, *Snappydata: A unified cluster for streaming, transactions and interactice analytics.*, in *CIDR*, 2017.

[61] J. Dittrich and A. Jindal, *Towards a one size fits all database architecture.*, in *CIDR*, pp. 195–198, 2011.

[62] "Vertica FlexStore." `https://www.vertica.com/2009/12/25/vertica-3-5-flexstore-the-next-generation-of-column-stores`, 2009.

[63] "IBM Informix Warehouse Accelerator." `http://www.iiug.org/library/ids_12/IWA%20White%20Paper-2013-03-21.pdf`, 2013.

[64] "MemSQL." `http://www.memsql.com/`, 2018.

[65] J. Schaffner, A. Bog, J. Krüger, and A. Zeier, *A hybrid row-column oltp database architecture for operational reporting*, in *Business Intelligence for the Real-Time Enterprise*, pp. 61–74. Springer, 2009.

[66] "Druid." `http://druid.io/`, 2018.

[67] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz, *Positional update handling in column stores*, in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 543–554, ACM, 2010.

[68] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, *et. al.*, *Db2 with blu acceleration: So much more than just a column store*, *Proceedings of the VLDB Endowment* **6** (2013), no. 11 1080–1091.

[69] Y. Sharma, P. Ajoux, P. Ang, D. Callies, A. Choudhary, L. Demailly, T. Fersch, L. A. Guz, A. Kotulski, S. Kulkarni, *et. al.*, *Wormhole: Reliable pub-sub to support geo-replicated internet services.*, in *NSDI*, vol. 15, pp. 351–366, 2015.

[70] "Scribe." `https://github.com/facebookarchive/scribe/wiki`, 2018.

[71] "Oracle Golden Gate." `http://www.oracle.com/technetwork/middleware/goldengate/overview/index.html`, 2018.

[72] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner, *Towards scalable real-time analytics: an architecture for scale-out of olxp workloads*, *Proceedings of the VLDB Endowment* **8** (2015), no. 12 1716–1727.

[73] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson, *Strongly consistent replication for a bargain*, in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pp. 52–63, IEEE, 2010.

[74] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris, *Middleware based data replication providing snapshot isolation*, in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 419–430, ACM, 2005.

[75] F. Akal, C. Türker, H.-J. Schek, Y. Breitbart, T. Grabs, and L. Veen, *Fine-grained replication and scheduling with freshness and correctness guarantees*, in *Proceedings of the 31st international conference on Very large data bases*, pp. 565–576, VLDB Endowment, 2005.

[76] R. Barber, M. Huras, G. Lohman, C. Mohan, R. Mueller, F. Özcan, H. Pirahesh, V. Raman, R. Sidle, O. Sidorkin, *et. al.*, *Wildfire: Concurrent blazing data ingest and analytics*, in *Proceedings of the 2016 International Conference on Management of Data*, pp. 2077–2080, ACM, 2016.

[77] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang, *Analytics in motion: High performance event-processing and real-time analytics in the same database*, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 251–264, ACM, 2015.

[78] "Lambda Architecture." `http://lambda-architecture.net/`, 2018.

[79] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey III, C. A. Soules, and A. Veitch, *Lazybase: trading freshness for performance in a scalable database*, in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 169–182, ACM, 2012.

[80] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, *Weaving relations for cache performance.*, in *VLDB*, vol. 1, pp. 169–180, 2001.

[81] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*, vol. 370. Addison-wesley New York, 1987.

[82] "Voldemort." `http://www.project-voldemort.com/`, 2018.

[83] "Couchbase." `http://www.couchbase.com/`, 2018.

[84] "Flock DB." `https://github.com/twitter/flockdb`, 2018.

[85] "Apache HBase." `http://hbase.apache.org/`, 2018.

[86] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, *Calvin: fast distributed transactions for partitioned database systems*, in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 1–12, ACM, 2012.

[87] J. M. Faleiro and D. J. Abadi, *Rethinking serializable multiversion concurrency control*, *Proceedings of the VLDB Endowment* **8** (2015), no. 11 1190–1201.

[88] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, *Logical physical clocks*, in *International Conference on Principles of Distributed Systems*, pp. 17–32, Springer, 2014.

[89] M. Patiño-Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso, *Middle-r: Consistent database replication at the middleware level*, *ACM Transactions on Computer Systems (TOCS)* **23** (2005), no. 4 375–423.

[90] "GRPC." `http://www.grpc.io/`, 2018.

[91] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd, *Existential consistency: measuring and understanding consistency at facebook*, in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 295–310, ACM, 2015.

[92] J. M. Smith, P. A. Bernstein, U. Dayal, N. Goodman, T. Landers, K. W. Lin, and E. Wong, *Multibase: integrating heterogeneous distributed database systems*, in *Proceedings of the May 4-7, 1981, national computer conference*, pp. 487–499, ACM, 1981.

[93] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, *et. al.*, *Towards heterogeneous multimedia information systems: The garlic approach*, in *Research Issues in Data Engineering, 1995: Distributed Object Management, Proceedings. RIDE-DOM'95. Fifth International Workshop on*, pp. 124–131, IEEE, 1995.

[94] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, *et. al.*, *A demonstration of the bigdawg polystore system*, *Proceedings of the VLDB Endowment* **8** (2015), no. 12 1908–1911.

[95] H. Garcia-Molina and K. Salem, *Sagas*, in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, vol. 16, pp. 249–259, ACM, 1987.

[96] K. Salem, H. Garcia-Molina, and J. Shands, *Altruistic locking*, *ACM Transactions on Database Systems (TODS)* **19** (1994), no. 1 117–165.

[97] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao, *Deuteronomy: Transaction support for cloud data.*, in *CIDR*, 2011.

[98] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz, *Overview of multidatabase transaction management*, in *Conference of the Centre for Advanced Studies on Collaborative research*, pp. 23–56, 1992.

[99] H. Garcia-Molina and B. Kogan, *Achieving high availability in distributed databases*, *IEEE Transactions on Software Engineering* **14** (1988), no. 7 886–896.

[100] H. K. Korth and G. Speegle, *Formal model of correctness without serializabilty*, vol. 17. ACM, 1988.

[101] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz, *Non-serializable executions in heterogeneous distributed database systems*, in *Parallel and Distributed Information Systems, 1991., Proceedings of the First International Conference on*, pp. 245–252, IEEE, 1991.

[102] W. Du and A. Elmagarmid, *Quasi serializability: a correctness criterion for global concurrency control in interbase*, in *Proceedings of the 15th International Conf. on Very Large Data Bases*, 1989.

[103] A. Dey, A. Fekete, and U. Röhm, *Scalable distributed transactions across heterogeneous stores*, in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pp. 125–136, IEEE, 2015.

[104] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, *Communications of the ACM* **21** (1978), no. 7 558–565.

[105] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, *Causal memory: Definitions, implementation, and programming*, *Distributed Computing* **9** (1995), no. 1 37–49.

[106] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, *Don't settle for eventual: scalable causal consistency for wide-area storage with cops*, in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 401–416, ACM, 2011.

[107] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, *Stronger semantics for low-latency geo-replicated storage.*, in *NSDI*, vol. 13, pp. 313–328, 2013.

[108] N. Viennot, M. Lécuyer, J. Bell, R. Geambasu, and J. Nieh, *Synapse: a microservices architecture for heterogeneous-database web applications*, in *Proceedings of the Tenth European Conference on Computer Systems*, p. 21, ACM, 2015.

[109] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, *Providing high availability using lazy replication*, *ACM Transactions on Computer Systems (TOCS)* **10** (1992), no. 4 360–391.

[110] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, *Bolt-on causal consistency*, in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 761–772, ACM, 2013.

[111] C. J. Fidge, *Timestamps in message-passing systems that preserve the partial ordering*, *Australian Computer Science Communications* **10** (1988), no. 1 56–66.

[112] F. Mattern *et. al.*, *Virtual time and global states of distributed systems*, *Parallel and Distributed Algorithms* **1** (1989), no. 23 215–226.

[113] "S Kimball. Living Without Atomic Clocks." `https://www.cockroachlabs.com/blog/living-without-atomic-clocks/`, 2016.

[114] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, *Building a database on s3*, in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 251–264, ACM, 2008.

[115] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, *Consistency-based service level agreements for cloud storage*, in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 309–324, ACM, 2013.

[116] S. Das, D. Agrawal, and A. El Abbadi, *G-store: a scalable data store for transactional multi key access in the cloud*, in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 163–174, ACM, 2010.

[117] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et. al.*, *Scaling memcache at facebook.*, in *nsdi*, vol. 13, pp. 385–398, 2013.

[118] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, *Workload analysis of a large-scale key-value store*, in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, pp. 53–64, ACM, 2012.

[119] B. M. Oki and B. H. Liskov, *Viewstamped replication: A new primary copy method to support highly-available distributed systems*, in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pp. 8–17, ACM, 1988.

[120] "CockroachDB Repository with Changes for Scaling Reads in Raft-like consensus protocols." `https://github.com/vaibhavarora/cockroach/tree/raft-read-scalability`, 2017.

[121] "RocksDB." `http://rocksdb.org/`, 2018.

[122] M. Maekawa, *An algorithm for mutual exclusion in decentralized systems*, *ACM Transactions on Computer Systems (TOCS)* **3** (1985), no. 2 145–159.

[123] D. Agrawal and A. El Abbadi, *The tree quorum protocol: An efficient approach for managing replicated data.*, in *VLDB*, vol. 90, pp. 243–254, 1990.

[124] "Scaling Raft: Multiraft." `https://www.cockroachlabs.com/blog/scaling-raft/`, 2018.

[125] "YCSB: Yahoo! Cloud System Benchmark." `https://github.com/brianfrankcooper/YCSB`, 2018.

[126] "Nmon for Linux: Nigel's performance Monitor for Linux." `http://nmon.sourceforge.net/`, 2018.

[127] A. E. Abbadi and S. Toueg, *Maintaining availability in partitioned replicated databases*, *ACM Transactions on Database Systems (TODS)* **14** (1989), no. 2 264–290.

[128] A. El Abbadi, D. Skeen, and F. Cristian, *An efficient, fault-tolerant protocol for replicated data management*, in *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pp. 215–229, ACM, 1985.

[129] T. D. Chandra, R. Griesemer, and J. Redstone, *Paxos made live: an engineering perspective*, in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pp. 398–407, ACM, 2007.

[130] I. Moraru, D. G. Andersen, and M. Kaminsky, *Paxos quorum leases: Fast reads without sacrificing writes*, in *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, pp. 1–13, ACM, 2014.

[131] F. P. Junqueira, B. C. Reed, and M. Serafini, *Zab: High-performance broadcast for primary-backup systems*, in *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pp. 245–256, IEEE, 2011.

[132] "Apache Zookeeper." `https://zookeeper.apache.org/`, 2018.

[133] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan, *Challenges to adopting stronger consistency at scale.*, in *HotOS*, 2015.

[134] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang, *Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases*, *Proceedings of the VLDB Endowment* **9** (2016), no. 6 504–515.

[135] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, *Tictoc: Time traveling optimistic concurrency control*, in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1629–1642, ACM, 2016.

[136] "CockroachDB Repository with Source-code Changes for Dynamic Commit Timestmap Allocation." `https://github.com/vaibhavarora/cockroach/tree/dynamic_timestamp_ordering`, 2018.

[137] "Benchmark for evaluating Dynamic Timestamp ordering based Concurrency Control in CockroachDB." `https://github.com/vaibhavarora/cockroach/tree/dynamic_timestamp_benchmark`, 2018.

[138] D. Agrawal and S. Sengupta, *Modular synchronization in multiversion databases: Version control and concurrency control*, in *Proceedings of the 2016 International Conference on Management of Data*, pp. 408–417, ACM, 1989.

[139] A. Fekete, E. O'Neil, and P. O'Neil, *A read-only transaction anomaly under snapshot isolation*, *ACM SIGMOD Record* **33** (2004), no. 3 12–14.

[140] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha, *Making snapshot isolation serializable*, *ACM Transactions on Database Systems (TODS)* **30** (2005), no. 2 492–528.

[141] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan, *Automating the detection of snapshot isolation anomalies*, in *Proceedings of the 33rd international conference on Very large data bases*, pp. 1263–1274, VLDB Endowment, 2007.

[142] M. J. Cahill, U. Röhm, and A. D. Fekete, *Serializable isolation for snapshot databases*, *ACM Transactions on Database Systems (TODS)* **34** (2009), no. 4 20.

[143] D. P. Reed, *Naming and synchronization in a decentralized computer system.* PhD thesis, Massachusetts Institute of Technology, 1978.

[144] M. Yabandeh and D. Gómez Ferro, *A critique of snapshot isolation*, in *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys)*, pp. 155–168, ACM, 2012.

[145] D. Agrawal and A. El Abbadi, *Locks with constrained sharing*, in *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 85–93, ACM, 1990.

[146] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, *Transaction chopping: Algorithms and performance studies*, *ACM TODS* **20** (1995), no. 3 325–363.

[147] C. Yan and A. Cheung, *Leveraging lock contention to improve oltp application performance*, *Proceedings of the VLDB Endowment* **9** (2016), no. 5 444–455.

[148] C. Boksenbaum, M. Cart, J. Ferrié, and J.-F. Pons, *Certification by intervals of timestamps in distributed database systems*, in *Proceedings of the 10th International Conference on Very Large Data Bases*, pp. 377–387, Morgan Kaufmann Publishers Inc., 1984.

[149] "CockroachDB Design." `https://github.com/cockroachdb/cockroach/blob/master/docs/design.md`, 2018.

[150] "Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020." `http://www.gartner.com/newsroom/id/2636073`, 2013.

[151] E. M. Schooler, D. Zage, J. Sedayao, H. Moustafa, A. Brown, and M. Ambrosin, *An architectural vision for a data-centric iot: Rethinking things, trust and clouds*, in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pp. 1717–1728, IEEE, 2017.

[152] "Apache Spark." `http://spark.apache.org/`, 2018.

[153] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W.-F. Wong, and M. Zhang, *Exploiting single-threaded model in multi-core in-memory systems*, *IEEE Transactions on Knowledge and Data Engineering (TKDE)* **28** (2016), no. 10 2635–2650.

[154] "Storm." `https://storm.incubator.apache.org/`, 2018.

[155] M. Lippi, M. Mamei, S. Mariani, and F. Zambonelli, *Coordinating distributed speaking objects*, in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pp. 1949–1960, IEEE, 2017.

[156] "Sensor Fusion." `http://www.mouser.com/applications/sensor-fusion-iot/`, 2018.

[157] I. Horrocks, M. Giese, E. Kharlamov, and A. Waaler, *Using semantic technology to tame the data variety challenge*, *IEEE Internet Computing* **20** (2016), no. 6 62–66.

[158] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, *Aurora: a new model and architecture for data stream management*, *the VLDB Journal* **12** (2003), no. 2 120–139.

[159] Z. Shen, V. Kumaran, M. J. Franklin, S. Krishnamurthy, A. Bhat, M. Kumar, R. Lerche, and K. Macpherson, *Csa: Streaming engine for internet of things.*, *IEEE Data Eng. Bull.* **38** (2015), no. 4 39–50.

[160] N. Tatbul, S. Zdonik, J. Meehan, C. Aslantas, M. Stonebraker, K. Tufte, C. Giossi, and H. Quach, *Handling shared, mutable state in stream processing with correctness guarantees.*, *IEEE Data Eng. Bull.* **38** (2015), no. 4 94–104.

[161] J. Gray and A. Reuter, *Transaction processing: concepts and techniques*. Elsevier, 1992.

[162] "Stored Procedures." `https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine`, 2018.

[163] R. Ramakrishnan and J. Gehrke, *Database management systems*. McGraw Hill, 2000.

[164] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, *et. al.*, *S-store: a streaming newsql system for big velocity applications*, *Proceedings of the VLDB Endowment* **7** (2014), no. 13 1633–1636.

[165] D. J. Abadi, S. Madden, and W. Lindner, *Reed: Robust, efficient filtering and event detection in sensor networks*, in *Proceedings of the 31st international conference on Very large data bases*, pp. 769–780, VLDB Endowment, 2005.

[166] D. Chu, A. Deshpande, J. M. Hellerstein, and W. Hong, *Approximate data collection in sensor networks using probabilistic models*, in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pp. 48–48, IEEE, 2006.

[167] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, *Tinydb: an acquisitional query processing system for sensor networks*, *ACM Transactions on database systems (TODS)* **30** (2005), no. 1 122–173.

[168] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong, *Model-driven data acquisition in sensor networks*, in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pp. 588–599, VLDB Endowment, 2004.

[169] I. Lazaridis and S. Mehrotra, *Capturing sensor-generated time series with quality guarantees*, in *Data Engineering, 2003. Proceedings. 19th International Conference on*, pp. 429–440, IEEE, 2003.

[170] S. Chaudhuri, *An overview of query optimization in relational systems*, in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 34–43, ACM, 1998.